Higher-Order Contracts for JavaScript

A dissertation submitted to the University of Freiburg in conformity with the requirements of the degree of Doctor of Natural Sciences (Dr. rer. nat.).

by

Roman Matthias Keil

University of Freiburg

Thesis Title
Higher-Order Contracts for JavaScript

Author
Roman Matthias Keil, M.Sc.

Supervisor of Dissertation
Prof. Dr. Peter Thiemann (University of Freiburg)

Dissertation Committee

Committee Chair	*Thesis Advisor*	*Thesis Reader*	*Thesis Reader*
Andreas Podelski	Peter Thiemann	Michael Pradel	Georg Lausen
University of Freiburg	University of Freiburg	TU Darmstadt	University of Freiburg

Dean
Prof. Dr. Oliver Paul (University of Freiburg)

Graduate School
University of Freiburg
Faculty of Engineering
Georges-Koehler-Allee, Building 101
D-79110 Freiburg i. Br., Germany

Date of Submission
March 31, 2017

Date of Disputation
March 8, 2018

Digital Object Identifier: 10.6094/UNIFR/16963

Contents

List of Figures

List of Tables

List of Listings

Abstract

JavaScript is an untyped and dynamic programming language with objects and first-class functions. While it is most well-known as the client-side scripting language for websites, it is also increasingly used for non-browser development, such as developing server-side applications with *Node.js*, for game development, to implement platform-independent mobile applications, or as a compilation target for other languages like *TypeScript* or *Dart*.

Unfortunately, JavaScript itself has no real security awareness: there is a global scope for functions and variables, all scripts have the same authority, and everything can be modified, from the fields and methods of an object over its prototype property to the scope chain of a function closure. As a consequence, JavaScript code is prone to injection attacks, library code can read and manipulate everything reachable from the global scope, and third-party code can get access to sensitive data. Furthermore, side effects may cause unexpected behavior so that program understanding and maintenance become difficult.

To overcome these limitations, we propose using contracts with runtime monitoring. Software contracts were introduced with Meyer's *Design by Contract*™ methodology which stipulates invariants for objects as well as Hoare-like pre- and postconditions for functions. Contract monitoring has become a prominent mechanism to provide strong guarantees for programs in dynamically typed languages while preserving their flexibility and expressiveness.

This dissertation presents the design and implementation of TreatJS, a language-embedded, higher-order contract system for JavaScript which enforces contracts by runtime monitoring. Beyond the standard abstractions for higher-order contracts (flat contracts, function contracts, dependent contracts), TreatJS provides intersection and union operators for contracts and a contract constructor that constructs and composes contracts at runtime using contract abstraction. Contract constructors are the building blocks for dependent contracts, parameterized contracts, and recursive contracts.

Another novel aspect is TreatJS's use of constraints to create a structure for computing positive and negative blame according to the semantics of subject and context satisfaction, respectively. Moreover, it applies a compatibility check to distinguish contracts from different sides of an intersection or union, and it provides three general monitoring semantics which handles the visibility of contracts inside of predicate code.

TreatJS is implemented as a library in JavaScript. It enables a developer to specify all aspects of a contract using the full JavaScript language. JavaScript proxies implement delayed contract checking of function and object contracts and guarantee full interposition for the full JavaScript language, including the `with`-statement and `eval`. Moreover, its implementation gives noninterference a high priority and it employs a membrane-based sandbox to keep the predicate code apart from the normal program execution.

Finally, the implementation of TreatJS illustrates the need for a different proxy constructor that is better suited for the implementation of contract wrappers. One issue with the current contract implementation arises because a contract wrapper is different (not pointer-equal) from its target object so that an equality test between wrapper and target returns false instead of true. Thus, TreatJS comes with an implementation of a transparent object proxy which ensures transparent operations with all JavaScript programs.

Zusammenfassung

JavaScript ist eine schwach typisierten und dynamische Programmiersprache mit Objekten und Funktionen erster Klasse. Obwohl JavaScript am meisten als clientseitige Skriptsprache für Webseiten bekannt ist wird sie mittlerweile auch vermehrt für nicht-Browser Entwicklungen eingesetzt, wie zum Beispiel für die Entwicklung von serverseitigen Anwendungen mit *Node.js*, für Spieleentwicklungen, für die Entwicklung von plattformunabhängigen Handy Applikationen, oder als Zwischensprache für andere Sprachen wie *TypeScript* oder *Dart*.

Leider besitzt JavaScript selbst nur ein geringes Sicherheitsbewusstsein: es gibt einen globalen Bereich für Funktionen und Variablen, alle Skripte haben die gleichen Rechte, und jeder kann alles verändern, von den Feldern und Methoden eines Objekts, über die Prototype-Eigenschaft, bis hin zu dem Sichtbarkeitsbereich von Variablen innerhalb einer Funktion. Die Konsequenz davon ist, dass JavaScript anfällig für Code-Injektion ist, Bibliotheken alles lesen können was über den globalen Bereich erreichbar ist und fremder Code Zugriff auf sensibel Daten bekommen kann. Darüberhinaus können unerwartete Seiteneffekte entstehen und Verständnis und Wartung von JavaScript Programmcode werden erschwert.

Um diesen Einschränkungen entgegenzuwirken schlagen wir die Verwendung von einem Vertragssystem mit Laufzeitüberwachung vor. Vertragssysteme wurden mit Meyer's *Design by Contract*™ Technologie eingeführt und sehen die Definition von Invarianten für Objekte, sowie Hoare-ähnliche Vor- und Nachbedingungen für Funktionen vor. Gerade für dynamisch typisierte Sprachen sind Vertragssysteme interessant da sie starke Programmgarantien ermöglichen ohne dabei die Flexibilität und Ausdrucksstärke der Sprache einzuschränken.

Diese Dissertation präsentiert Design und Implementierung von TreatJS, einem Vertragssystem höherer Ordnung für JavaScript. Neben den Standard Abstraktionen für Verträge stellt TreatJS Schnitt- und Vereinigungsoperatoren für Verträge und einen Vertragskonstruktor, welcher Verträge mittels Abstraktion zur Laufzeit erzeugt, zur Verfügung. Vertragskonstruktoren sind die Bauelemente für alle zustandsbehafteten Vertrage, wie zum Beispiel argumentabängige Verträge, parametrierbare Verträge oder rekursive Verträge.

Eine weitere Neuerung in TreatJS ist die Verwendung von Constraints zur Berechnung von Schuldzuweisungen nach Vertragsverletzungen. Des Weiteren verwendet TreatJS eine Kompatibilitätsprüfung um Verträge von unterschiedlichen Seiten eines Schnitt- oder Vereinigungsoperator auseinanderhalten und TreatJS stellt drei verallgemeinerte Semantiken zur Verfügung, welche die Sichtbarkeit von Verträgen in Prädikaten regeln.

TreatJS ist als eine Bibliothek in JavaScript entwickeln. Die Bibliothek ermöglicht es einem Entwickler alle Aspekte eines Vertrags in JavaScript selbst zu spezifizieren. JavaScript Proxies übernehmen die Überprüfung von Objekt und Funktionsverträgen zur Laufzeit und garantieren eine vollständige Zwischenschaltung der Verträge für die komplette JavaScript Sprache, inklusive des `with` Statements und `eval`. Darüberhinaus räumt TreatJS Nichteinmischung eine hohe Bedeutung ein und stellt eine eingebettete Sandbox zur Verfügung um Prädikate in Isolation zur normalen Programmausführung zu überprüfen.

Darüberhinaus zeigt die Implementierung von TreatJS die Notwendigkeit für einen anderen Proxy Konstruktor welche besser für die Entwicklung eines Vertragssystems geeignet ist. Ein Problem mit der aktuellen Implementierung in JavaScript ist, dass Proxy Objekte nicht referenzgleich zu ihrem Zielobjekt sind, und daher jeder Vergleich zwischen Proxy und Zielobjekt zu einem veränderten Ergebnis führt. Aus diesem Grund stellt TreatJS auch die Implementierung von einem transparenten Proxy in der Virtuellen Maschine zur Verfügung.

Preface

This dissertation is submitted to the University of Freiburg in conformity with the requirements of the degree of Doctor of Natural Sciences (Dr. rer. nat.). It contains work done between May 2011 and March 2017 at the University of Freiburg, Institute for Computer Science, while working in the group Progλang under the supervision of Prof. Dr. Peter Thiemann.

Prior Publications

This dissertation incorporates material that is based on the following publications by the author of this thesis and others:

- Matthias Keil, 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

- Matthias Keil, Peter Thiemann
  TreatJS: Higher-Order Contracts for JavaScripts
  In Proceedings of the European Conference on Object-Oriented Programming, ECOOP 2015

- Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, Peter Thiemann
  Transparent Object Proxies for JavaScript
  In Proceedings of the European Conference on Object-Oriented Programming, ECOOP 2015

- Matthias Keil, Peter Thiemann
  Efficient Dynamic Access Analysis Using JavaScript Proxies
  In Proceedings of the Dynamic Languages Symposium, DLS 2013

The following publications contain material that has been adapted to this thesis.

- Matthias Keil, Peter Thiemann
  Static Contract Simplification (Technical Report)
  Institute for Computer Science, University of Freiburg, 2017

- Matthias Keil, Peter Thiemann
  Transaction-based Sandboxing for JavaScript (Technical Report)
  Institute for Computer Science, University of Freiburg, 2016

- Matthias Keil, Omer Farooq, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, Peter Thiemann
  Transparent Object Proxies for JavaScript (Extended Abstract)
  In Proceedings of the Software Engineering-Konferenz, SE 2016

- Matthias Keil, Peter Thiemann
  On Contracts and Sandboxes for JavaScript
  In Proceedings of the 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015

- Matthias Keil, Peter Thiemann
  TreatJS: Higher-Order Contracts for JavaScripts (Technical Report)
  Institute for Computer Science, University of Freiburg, 2015

- Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, Peter Thiemann
  Transparent Object Proxies for JavaScript (Technical Report)
  Institute for Computer Science, University of Freiburg, 2015

- Matthias Keil, Peter Thiemann
  On the Proxy Identity Crisis (Position Paper)
  Institute for Computer Science, University of Freiburg, 2013

- Matthias Keil, Peter Thiemann
  Efficient Dynamic Access Analysis Using JavaScript Proxies (Technical Report)
  Institute for Computer Science, University of Freiburg, 2013

Furthermore, this dissertation presents the design and implementation of the following accepted artifacts.

- Matthias Keil, Peter Thiemann
  TreatJS: Higher-Order Contracts for JavaScripts
  The European Conference on Object-Oriented Programming, ECOOP 2015 Artifacts

- Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, Peter Thiemann
  Transparent Object Proxies for JavaScript
  The European Conference on Object-Oriented Programming, ECOOP 2015 Artifacts

Finally, the following publications by the author of this thesis and others emanated from the scientific research on this topic.

- Matthias Keil, Peter Thiemann
  Symbolic Solving of Extended Regular Expression Inequalities
  In Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2014

- Matthias Keil, Peter Thiemann
  Symbolic Solving of Extended Regular Expression Inequalities (Technical Report)
  Institute for Computer Science, University of Freiburg, 2014

- Matthias Keil, Peter Thiemann
  Type-based Dependency Analysis for JavaScript

In Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS'13

- Matthias Keil, Peter Thiemann
  Type-based Dependency Analysis for JavaScript (Technical Report)
  Institute for Computer Science, University of Freiburg, 2013

Acknowledgments

I would like to thank my supervisor, Prof. Dr. Peter Thiemann, for his guidance and support in research matters, for a lot of great inspirations, and for the opportunity to do my Ph.D. in his group. Besides my supervisor, I would like to thank Sam Tobin-Hochstadt for serving as a reviewer for my thesis and the rest of my dissertation committee, Michael Pradel, Andreas Podelski, and Georg Lausen, for spending their time and participating in the disputation.

I would also like to thank my colleagues at the Progλang research group for their constant support and friendship, in particular to Robert Jakob, Manuel Geffken, and Luminous Fennell. I am grateful to have been part of the Progλang research group and to work with wonderful people in a great atmosphere.

Thanks also to all the people I met at conferences, seminars, and workshops for a lot of informative discussions and for stimulating plenty of ideas – some of which are now part of the work. In particular, Tom Van Cutsem provided helpful advice on the internals of JavaScript proxies. Many thanks also to Jack Williams, Philip Wadler, Robby Findler, and Sam Tobin-Hochstadt for insightful discussions on intersection and union contracts and Mark S. Miller and Philip Wadler for a substantive debate on transparent proxies.

Thanks also to Dominik Erb for his time and all the spirited discussions at lunch. To Christian Ortolf for his support and friendship during my study. To my former group members, Dr. Phillip Heidegger, and Dr. Annette Bieniusa, for introducing me to research and academic work. To Marlis Jost, Berit Brauer and Martin Preen for their administrative and technical support. And to my student assistants Omer Farooq, Sankha Narayan Guria, Andreas Schlegel for implementing prototypes and for their enthusiasm for my work. Lots of thanks are also due to Johannes Wangler, Luminous Fennell, Dominik Erb, and Christian Ortolf for expertly proofreading this thesis and a lot of helpful comments.

Finally, I would like to thank my family and friends for their unfailing encouragement, there patient, and for being helpful and supportive during the years of my work.

Roman Matthias Keil
Freiburg i. Br., October 1, 2018

»Philosophers are people who know less and less about more and more, until they know nothing about everything. Scientists are people who know more and more about less and less, until they know everything about nothing.«

— *Konrad Lorenz*

1 Introduction

JavaScript [33] is an untyped and dynamic programming language with objects and first-class functions. While it is most well-known as the client-side scripting language for web sites[1], it is also increasingly used for non-browser development, such as developing server-side applications with *Node.js* [84], for game development, to implement platform-independent mobile applications, or as an intermediate language for other languages to target, such as *TypeScript* [106] or *Dart* [22]. Hence, it does not come as a surprise that JavaScript is now in the focus of many researchers' works, out of the need to create better development tools for JavaScript programmers and to launch new language features.

Syntax and semantics of JavaScript are standardized in the *ECMAScript* language specification [33]. The specification includes the grammatical structure of JavaScript along with the semantics of that structure and a formal specification of JavaScript's core API. Even though JavaScript itself provides only a small core API, it includes built-in objects for representing numbers and dates, mathematical calculations, text processing with regular expressions, and to describe collections of data. It also includes a number of fundamental objects, upon which all other objects are based. In contrast to other languages, JavaScript's core API does not contain any I/O facilities like file reading, networking, event handling, or graphics. Providing features like this is up to the host environment. The most well-known host extension is the *Document Object Model (DOM)*, an API for manipulating HTML and XML documents provided by web browsers.

JavaScript is a platform-independent language, typically implemented as an interpreted language. There exists a large number of active JavaScript implementations (aka JavaScript engines). The most common engines are Mozilla's *SpiderMonkey* [99] engine, which is part of the *Gecko* (*Firefox*) project, Google's *V8* [107] JavaScript engine, which is used in *Google Chrome* and *Node.js*, Microsoft's *Chakra* [14] for *Microsoft Edge*, or *JavaScriptCore* [59] (*Webkit*), which is marketed as Nitro and used on Apple devices.

Even though JavaScript is an interpreted language, most engines apply just-in-time (JIT) compilation or variations of that technology when executing JavaScript programs to improve the performance of applications written in JavaScript. It is this fact that makes JavaScript a popular and ubiquitous programming language. JavaScript is the most popular and fastest growing client-side programming languages on the web.

JavaScript Issues

Almost all JavaScript programs rely on third-party libraries, such as *AngularJS*, *jQuery*, or *Prototype*, to extend the native API with new features and to help developers to concentrate upon more distinctive applications. Some of these libraries are packed with the application, but others are loaded at runtime. Hence, the finally executed code of a JavaScript program may be composed of scripts from different origins, sometimes accumulated by dynamic loading and fragments from runtime code generation.

Unfortunately, JavaScript itself has no real security awareness: there is no namespace or encapsulation management, there is a global scope for functions and variables, all scripts have the same authority, and everything can be modified, from the fields and methods of an object over its prototype property to the scope chain of a function closure.

[1] 94.5% of all websites use JavaScript, according to `http://w3techs.com` (status of March 2017)

As a consequence, JavaScript code is prone to injection attacks. Library code can read and manipulate everything reachable from the global scope and third-party code can get access to sensitive data. Furthermore, side effects may cause unexpected behavior and program understanding and maintenance becomes difficult.

Facebook [34] is one of the best-known examples for a website that makes substantial use of JavaScript. The social network stores and displays private data of their users, allows to have public conversations and to send private messages, shows user specific advertisement, and allows their users to add their own application to the main site.

Facebook applications are intended to interact with the user's profile. They gain access to the Facebook page document and does not run in isolation from the main site. However, their action has to be restricted. The application should neither manipulate anything but its own realm nor should it communicate or perform unauthorized actions on behalf of the user. Thus, applications have to be written in an HTML variant and a JavaScript subset that allows to reason about its behavior.

Key Challenges of Present Research on JavaScript

Today's state of the art in securing JavaScript applications that include code from different origins is an all-or-nothing choice: scripts either run in isolation or gain full integration. Browsers apply protection mechanisms such as the *same-origin policy* [93] or the *signed script policy* [98], both of which decide whether to grant access to a particular resource or not.

Clearly, script isolation is the first choice. It guarantees noninterference with the working of the application as well as the preservation of data integrity and confidentiality. However, some scripts must have access to the application state and others are even allowed to change it while preserving the integrity and confidentiality constraints of the host application.

However, all included scripts run in combination with the main script in the user's browser and the host application cannot exert control over the included libraries. As some JavaScript fragments are ill-behaved, it requires to control the use of data by included scripts, to investigate effects, and to apply policies that restrict the behavior of a program.

This sounds like a challenge to static analysis techniques, to compilers, to automated testing, and to type systems, all of which inspect the code of a program to make guarantees about the runtime behavior of that program. However, because of JavaScript's dynamics, there is very little a traditional language-based analysis mechanism can do.

Thus, managing untrusted JavaScript code has become one of the key challenges of present research on JavaScript [2, 53, 23, 24, 92, 77, 88, 75, 74, 48]. Existing approaches are either based on restricting JavaScript code to a statically verifiable language subset (e.g., Facebook's FBJS [35] or Yahoo's ADsafe [1]) or on enforcing an execution model that only forwards selected resources into an otherwise isolated compartment by filtering and rewriting like Google's Caja project [47, 79]. However, these approaches have known deficiencies: They either need to restrict the usage of JavaScript's dynamic features or they do not apply to code generated at runtime. Furthermore, they require extra maintenance efforts because their analysis needs to be adapted when the language evolves.

Contracts with Run-Time Monitoring

One possible solution is to have contracts with runtime monitoring. Software contracts were introduced with Meyer's *Design by Contract*™ methodology [76] which stipulates invariants for objects as well as Hoare-like pre- and postconditions for functions a programmer regards as essential for the correct execution of a program. Since then, the contract idea has taken

off and attracted a plethora of follow-up works that range from contract monitoring for higher-order functional languages [41] over semantic investigations [9, 40] and studies on blame assignment [26, 108] to extensions in various directions: polymorphic contracts [3, 8], behavioral and temporal contracts [8, 32], etc.

Contract monitoring has become a prominent mechanism to provide strong guarantees for programs in dynamically typed languages while preserving their flexibility and expressiveness. Hence, it does not come as a surprise that the first higher-order contract systems were devised for Scheme and Racket [41], out of the need to create maintainable software. Other dynamic languages like JavaScript[61, 17], Python [90], PHP [87], Ruby [18], and Lua [73] have followed suit.

Nowadays, contract systems are available for many languages [41, 57, 62, 70, 55, 16, 36, 13] and come with a wealth of features [64, 54, 8, 32, 105, 28, 3].

Language-embedded Systems

Unlike static verification methods, which are imprecise due to JavaScript's dynamic features, dynamic monitoring guarantees full observability and works for all code, regardless of its origin. However, implementing monitoring facilities inside a JavaScript engine (like it is done for WebKit [92]) is fragile and incomplete, as such a solution only works for one engine and is hard to maintain due to the high activity in engine development and optimization.

Thus, many contract systems [57, 16, 36, 64, 105, 28, 3] are *language-embedded*: They are implemented as a library in the target language itself and all aspects are accessible through an API. Language-embedded systems are distributed as a language extension and can easily be included in existing projects. No source code transformation or change in the JavaScript runtime system is required.

This approach is advantageous because it does not tie the contract system to a particular implementation, users do not need to learn a separate contract language, and there is no need to have specialized contract tools.

1.1 Contributions of this Thesis

This dissertation presents TreatJS, a language-embedded, higher-order contract system for JavaScript which enforces contracts by runtime monitoring. Beyond providing the standard abstraction for building higher-order contracts (flat, function, and object contracts), TreatJS's novel contributions are its systematic approach to blame assignment, its support for contracts in the style of intersection and union types, its guarantee of a non-interfering contract execution, and its notion of contract abstraction, which is the building block for dependent contracts, parameterized contracts, stateful contracts, and recursive contracts.

The content of this dissertation encompasses the following research topics.

1.1.1 Higher-order Contracts for JavaScript

The main part of this thesis focuses on the design and implementation of TreatJS. TreatJS supports most features of existing contemporary contract systems (embedded contract language, contracts as projections, full interposition) in combination with a wealth of novel features and features that have not been implemented in this combination before.

TreatJS is implemented as a library in JavaScript and relies on JavaScript proxies to guarantee full interposition of contracts. It further exploits reflection and JavaScript's dynamic features to run contract code in a configurable degree of isolation, which guarantees

that the execution of contract code does not interfere with the execution of a contract-abiding host program.

Parts of this work appeared in the conference proceedings of the *European Conference on Object-Oriented Programming, ECOOP 2015* [68] and in the proceedings of the *20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015* [67].

1.1.2 Static Contract Simplification

Contracts and contract monitoring are a powerful mechanism for enforcing guarantees at runtime. However, the insertion of contract checks and the introduction of proxy objects significantly impact the execution time of a program.

To overcome this issue, this part presents *static contract simplification*. Its objective is to pre-evaluate contracts at compile time and to apply compile-time transformations to reduce as much from a contract as possible. Our key technique is to propagate contracts statically through the program code and to evaluate contracts where possible. Remaining fragments, which cannot be checked statically, where lifted to the enclosing boundary and condensed to shorter interface descriptions which are collectively cheaper to check at runtime.

1.1.3 Transaction-based Sandboxing for JavaScript

The ideal contract system should not interfere with the execution of a contract-abiding host application as long as the application code does not violate any contract. Interference issues between the host application and contract monitoring arise from executing unrestricted JavaScript code in predicates. This code may try to write to an object that is also visible to the application, it may throw an exception, or it may not terminate. TreatJS forbids all write operations to external objects using sandboxing.

This part presents DecentJS, a language-embedded sandbox for full JavaScript. It enables scripts to run in a configurable degree of isolation with fine-grained access control. It further provides a transactional scope in which effects are logged for inspection and in which effects can be committed to the application state or rolled back.

The implementation relies on JavaScript proxies to guarantee full interposition for the full JavaScript language and for all code, including dynamically loaded scripts and code injected via `eval`. Its only restriction is that scripts must be compliant with JavaScript's strict mode.

1.1.4 Transparent Object-Proxies for JavaScript

A proxy is a variant of an object that mediates access to another target object. A typical use case is contract monitoring, where proxies implement contracts on target objects. The proxy is then intended to be used in place of the target.

However, JavaScript proxies come with a limitation: A proxy, wrapping a target object, is a *new* object and different from its target. For distinct proxies, the equality (`==`) and strict equality (`===`) operator returns false, even if the target object is the same.

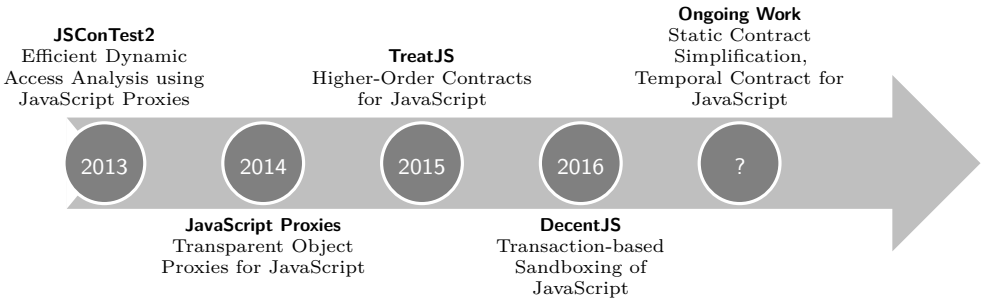
Object equality becomes an issue if a proxy is different (i.e., not pointer-equal) from the wrapped target object and an equality test between proxy and target object (or between two proxy objects of the same target) return false instead of true. More precisely, the expected result of such an equality test depends on the use case.

Thus, one important question in the design of a proxy API is whether a proxy object should inherit the identity of its target. Apparently, proxies should have their own identity for security-related applications whereas other applications, in particular, contract systems, require transparent proxies that compare equal to their target objects.

In this part, we show that a significant number of object comparisons fail when mixing opaque proxies and their target objects, e.g., when gradually adding contracts to a program. As neither the transparent nor the opaque implementation of proxies is appropriate for all use cases, we propose an alternative design for *transparent proxies* that is better suited for use cases like contract wrappers and access restricting membranes. The new proxy object is transparent with respect to equality. However, we use object capabilities to create proxies in an identity-realm and create an equality function that reveals proxies of its realm.

Parts of this work appeared in the conference proceedings of the *European Conference on Object-Oriented Programming, ECOOP 2015* [63].

1.1.5 Timeline



Our work on contracts started with the implementation of JSConTest2, which is a redesign and a reimplementation of JSConTest [54] using JavaScript proxies. JSConTest2 is a framework that helps to investigate effects of unfamiliar JavaScript code by monitoring read and write operations on objects through *access permission contracts*.

The proxy-based implementation addresses some shortcomings of the previous, translation-based, version: it guarantees full interposition for the full JavaScript language, it runs faster and consumes less memory, and it is safe for future language extensions. But the proxy-based implementation also comes with a limitation. For distinct proxies the normal (==) and strict (===) equality operator returns false, even if the target object is the same. So, out of the need to implement non-interfering contract monitors, we start with examining transparency issues with proxy objects in JavaScript. To overcome this, we provide an implementation of a new transparent proxy constructor that is better suited to implement contract wrappers.

Our main project, TreatJS, is a further development of JSConTest2. Both are language-embedded systems and both built on the same technological basis, JavaScript proxies and membranes, to implement dynamic contract monitoring. Whereas JSConTest2 only applies access permission contracts, TreatJS applies higher-order contracts in the style of Findler and Felleisen [41]. Furthermore, TreatJS applies a membrane-based sandbox to encapsulate the execution of the predicate code and to guarantee a non-interfering contract execution.

Later, this sandbox (DecentJS) has become a standalone project. DecentJS enables to execute JavaScript functions in isolation to the application state and it provides a transactional scope in which effects are logged for inspection by an access control policy.

Our ongoing work focusses on compile-time contract simplification and further contract constructs, like temporal contracts for JavaScript, polymorphic contracts, stateless and stateful contracts, and the adaptation of other constructs known from type systems.

1.2 Outline of this Thesis

This dissertation is structured into four main parts which are organized as follows: Chapter 3 starts with an introduction into object reflection and JavaScript proxies.

Part I Chapter 4 introduces TreatJS from a programmer's point of view and illustrates the basic principles underlying the contract system through a series of examples. Chapter 5 explains how contract monitoring in TreatJS works. Chapter 6 explains the principles underlying the implementation and Chapter 7 reports on our experiences from applying TreatJS to a range of benchmark programs.

Part II Chapter 8 introduces DecentJS, the language-embedded sandbox used in TreatJS. Chapter 9 provides a series of examples that explain DecentJS's facilities. Chapter 10 presents the design principles underlying DecentJS and Chapter 11 reports on our experience with applying DecentJS to benchmark programs.

Part III Chapter 12 discusses different use cases of JavaScript proxies and assesses them with respect to the requirements on proxy transparency. Chapter 13 discussion the programmer's expectation from an equality operation and presents alternative design to obtain proxy transparency. Chapter 14 presents a novel design for transparent proxies in the VM and Chapter 15 sketches the prototype implementation of an observer proxy, a variation of a transparent proxy that implements a projection.

Part IV Chapter 16 introduces the basic ideas underlying our ongoing work on static contract simplification. Chapter 17 explains the simplification through a series of examples and Chapter 18 gives an insight into the runtime improvements of our contract simplification.

Finally, Chapter 19 discusses related work not already mentioned in the thesis, Chapter 20 addresses some open research challenges concerning TreatJS, and Chapter 21 concludes.

2 On JavaScript

This chapter introduces some *JavaScript* features which are essential for this thesis. This chapter is not to introduce JavaScript's syntax and semantics as a whole; it's purpose is only to recap some language-concepts which are special to the language and which might be unusual and confusing for non-JavaScript developers.

The `eval` Function

Let's start with the most prominent example: `eval`. The `eval` function takes a string value representing a JavaScript expression, statement, or sequence of statements, and evaluates the string as normal JavaScript code in the current execution environment. The following example demonstrates using `eval` for evaluating an arithmetic expression.

```
1 eval("1 + 1"); // returns 2
```

Listing 2.1 Example of using `eval`.

While evaluating the JavaScript code, `eval` can access variables and functions defined in the enclosing scope, for example, it can access the variable `x` for evaluating an expression.

```
1 let x = 1:
2 eval("x + 1"); // returns 2
```

Listing 2.2 Example of using `eval` (cont'd).

Moreover, `eval` can override existing variables and it can declare new variables and functions in the enclosing scope. The following example demonstrate the creation of a `plus` function within `eval` which can be accessed from the current execution environment.

```
1 eval("let plus = function(x, y) { return x+y; }"); // returns undefined
2 plus(1, 1); // returns 2
```

Listing 2.3 Example of using `eval` (cont'd).

The example above creates a new function, `plus`, in the surrounding context which can later be used by other JavaScript code.

Indirect uses of `eval`

Apart from directly calling `eval`, JavaScript enables developers to call `eval` indirectly, i.e., by invoking it via a reference other than `eval`. Unlike the direct use of `eval` which evaluates the JavaScript code in the local scope, indirect uses of `eval` evaluate code in the global scope. This means that `eval` can be used to access and declare global variables and functions, as the following example demonstrates.

```
1 let e = eval;
2 e("let plus = function(x, y) { return x+y; }"); // returns undefined
```

Listing 2.4 Example of using `eval` (cont'd).

This concept enables JavaScript developers to extend the global scope with new functions, but also to manipulate and override anything reachable from the global scope.

Strict Mode

JavaScript's *strict mode* is an optional restricted variant of JavaScript. However, strict mode is not a subset of the normal JavaScript semantics as it provides a slightly different semantics.

Strict mode can either be applied to the entire script or to individual functions. To invoke strict mode, developers can put the `"use strict";` statement in the top of a JavaScript script or the top of a function body.

```
"use strict";
let x = 1 + 1;
```

Listing 2.5 Example of using JavaScript's strict mode.

One important change in JavaScript's strict mode semantics is that `eval` can no longer introduce new variables in the surrounding scope. Usually, `eval` can access and declare new variables and functions in the surrounding scope. However, in strict mode `eval` van only access and overwrite existing variables, but not declare new ones.

To invoke script mode for `eval`, one can either invoke `eval` from a strict mode context or place the `"use strict"` statement in front of the string.

```
eval("let x = 1 + 1; x;"); // returns 2
x; // returns undefined
```

Listing 2.6 Example of using JavaScript's strict mode (cont'd).

JavaScript's `with` Statement

The ***`with`*** *statement* adds an object to the head of the scope chain used while evaluating the statement's body. As JavaScript's property lookup searches for unqualified property names in the enclosing scopes, the `with` statements extends the scope with the properties defined in the object. If a variable is not defined in the local scope, then the variable access ends up in a property lookup on the given object, as the following example demonstrates.

```
with({ x:1, y:1 }) {
  let r = x+y; // returns 2
}
```

Listing 2.7 Example of using JavaScript's `with` statement.

Using `with` is very special as it is subject to a certain kind of dynamic: as the properties in the object can change over time, we can extend or change the scope of the nested statements. In general, scopes can only be changed from the inside, i.e., from scripts that run in that scope. The `with` statement, in contrast, enables developers to extend or change the scope by changing the object, which can also be done from another scope.

```
let object = { x:1, y:1 };
with(object) {
  let r = x+y; // sets r to 2
}
object.x = 2;
with(object) {
  let r = x+y; // sets r to 3
}
```

Listing 2.8 Example of using JavaScript's `with` statement (cont'd).

Moreover, the `with` statement enables developers to place an object with a getter or setter function in top of the scope chain and thus to enhance variables with its own state, encoded in the scope of the getter or setter function.

```
let i = 1;
let object = { get x() { return i++; }, get y() { return i; } };
with(object) {
  let r = x+y; // sets r to 2
}
```

Listing 2.9 Example of using JavaScript's `with` statement (cont'd).

Unqualified this Pointer

JavaScript's *`this` keyword* usually refers to the receiver of a function. However, this depends on how the function is called and whether we are in strict-mode or normal-mode. In the global context, `this` refers to the global object. In a function context, the value of `this` depends on how the function is called.

In non-strict mode and when calling a function directly without giving a value for `this`, `this` will point to the global object.

```
function f() {
  return this;
}
f(); // returns the global object
```

Listing 2.10 Example of using JavaScript's `this` keyword.

In strict-mode, in contract, `this` will refer to whatever it was set to before entering the function, or undefined if it was not set before.

```
function f() {
  return this;
}
f(); // returns undefined
```

Listing 2.11 Example of using JavaScript's `this` keyword (cont'd).

However, when calling a function as a method on an object, or when using methods like `call`, `apply`, and `bind` which define an explicit value for `this`, then `this` refers to that value, whether in strict mode or not. The following example demonstrates this behavior.

```
let object = {};
object.f = function () {
  return this;
}
object.f(); // returns object
```

Listing 2.12 Example of using JavaScript's `this` keyword (cont'd).

```
let object = {};
function f() {
  return this;
}
f.call(object); // returns object
```

Listing 2.13 Example of using JavaScript's `this` keyword (cont'd).

Unqualified `this` pointers typically arise when calling a function without defining an explicit value for `this`. This concept can be used to always obtain a pointer to the global object, simply by calling a function directly and returning the `this` value.

```
1 let global = (function() { return this; })();
```

Listing 2.14 Example of using JavaScript's `this` keyword (cont'd).

As this construct can be used in any context, each execution context can access the global object, even it is nested somewhere in another execution context. However, JavaScript's strict-mode prohibits unqualified `this` pointers as it requires `this` to be defined explicitly or it returns undefined.

Redefining Global Values

Another particularity of JavaScript is that each script can access and override anything reachable from its own or the global execution context. This also includes global and native functions and methods.

```
1 Function.prototype.toString = (function() { return "Hallo"; })();
```

Listing 2.15 Example of overriding JavaScript's `Function.prototype.toString` method.

In this example, the included script overrides the global `Function.prototype.toString` method. Each subsequent call of the `toString` methods, either as a method call on a function object using `call` or `apply` uses the new function instead of the original one.

```
1 function plus(x, y) {
2   return x+y;
3 }
4 plus.toString(); // return  "Hallo"
```

Listing 2.16 Example of overriding JavaScrip's `Function.prototype.toString` method (cont'd).

This feature makes it pretty hard to reason about the effects of JavaScript function as we can never be sure which function we execute. Guarantees can only be given if a library is loaded first of all and stores itself a reference to the original functions it attempts to use.

3 Object Reflection in JavaScript

Reflection is a meta-programming technique that enables a programming language to examine and to modify its own structure or behavior during program execution. Reflection is essential to build language-embedded systems that monitor program code while the program executes. Thus, reflection is commonly used for dynamic program analysis, to instantiate mock objects, or to enforce certain properties at runtime. Formally, meta-programming distinguishes three kinds of mediation [12, 82, 69]:

Introspection. Introspection is the ability to examine and to reason about the program's own state. A meta-program gains read-only access to a model of itself.

Self-modification. Self-modification is the ability to modify its own structure. A meta-program gains write access to a model of itself.

Intercession. Intercession is the ability to redefine its own semantics. A meta-program can modify its own execution state or alter its own behavior.

JavaScript provides reflection facilities by two built-in objects: `Reflect` and `Proxy`. Whereas the `Reflect` object provides methods to delegate interceptable operations to the default implementation of these operations, the `Proxy` object enables to enhance the functionality of a target object and to fully interpose all operations on an object, including property lookup, property assignment, property enumeration, and functions call on a function object.

This chapter introduces the JavaScript Proxy API, the new meta-programming features of the ECMAScript 6 [33] standard. JavaScript proxies have already been used to implement revocable references [21], for client-side sandboxing of third-party JavaScript code [2], for Disney's JavaScript contract system *Contracts.js* [30], to enforce access permission contracts [64], and as cross-compartment wrappers in SpiderMonkey's compartment concept [109].

3.1 Proxies

A *proxy* is an object intended to be used in place of another *target object*, which may be a *native object* (a *non-proxy object*) or another proxy object. As a proxy may be the target for another proxy, we call the native object that is transitively reachable through a chain of proxies the *base target* for each proxy in this chain. The behavior of a proxy is controlled by a *handler object*, which may modify the original behavior of the target object in many respects. A typical use case is to have the handler to mediate access to the target object.

The JavaScript Proxy API [33] provides a `Proxy` constructor that takes the proxy's target object and a handler object. Both target and handler may be in turn a proxy object.

```
1 let target = { /* some target object */ };
2 let handler = { /* some handler object */ };
3 let proxy = new Proxy (target, handler);
```

Listing 3.1 Example of a proxy construction.

The handler object is a placeholder which provides optional *trap methods* that are called when the corresponding operation is performed on the proxy. Operations like property read, property assignment, and function application result in a meta-level call to the corresponding trap in the handler object. The trap function may implement the operation arbitrarily, for example, by forwarding the operation to the target object. The latter is the default functionality if the trap is not specified.

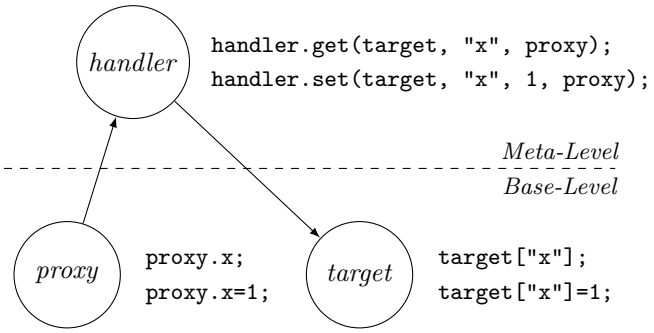


Figure 3.1 Example of a proxy operation. The property get `proxy.x` invokes the trap `handler.get(target, "x", proxy)` and the property set operation `proxy.x=1` invokes trap `handler.set(target, "x", 1, proxy)`. The handler simply forwards the operation to the proxy's target.

The following listing demonstrates the implementation of a default handler which forwards all property `get` and `set` operations to their default implementation.

```
var handler = {
  get: function(target, name, receiver) {
    return Reflect.get(target, name, receiver);
  },
  set: function(target, name, value, receiver) {
    return Reflect.set(target, name, value, receiver);
  }
}
```

Listing 3.2 Example of a proxy handler.

For example, a property get like `proxy.x` invokes the trap `handler.get(target, "x", proxy)` if that trap is present and a property assignment like `proxy.x=1` may invoke `handler.set(target, "x", 1, proxy)`. Untrapped operations are forwarded to the `target` object by default. Figure 3.1 illustrates this situation with a handler that simply forwards all operations. Performing an operation (like property get or property set) on the proxy object results in a meta-level call to the corresponding trap on the handler object.

However, a handler may redefine or extend the semantics of an operation arbitrarily. For example, a handler may implement a copy-on-write policy on its target object by intercepting all property write operations and serving reads on them locally. The following listing demonstrates the implementation of such a handler object.

```
let handler = (function() {
  const local = {};
  return {
    get: function(target, name, receiver) {
      return Reflect.get((name in local) ? local : target, name, receiver);
    },
    set: function(target, name, value, receiver) {
      return Reflect.set(local, name, value, receiver);
    }
  }
})();
```

Listing 3.3 Example of a proxy handler that implements a copy-on-write policy.

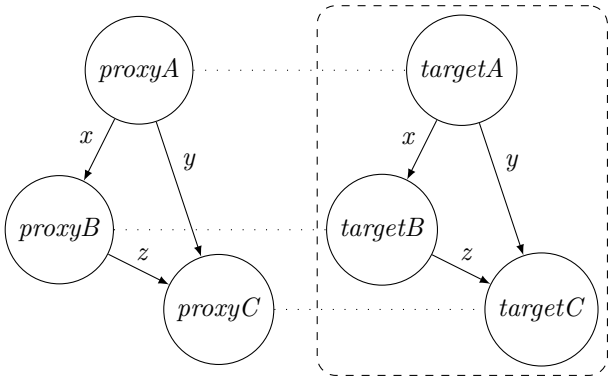


Figure 3.2 Property access through an identity preserving membrane. The property access through the wrapper `proxyA.x` returns a wrapper for `targetA.x`. The property access `proxyA.y` returns the same wrapper as `proxyB.z`.

The handler forwards the access to its target object only if the property is not locally present. Thus, reading a property from the proxy may return a value different than reading the property from the target.

3.2 Membranes

A *membrane* is a regulated communication channel between an object and the rest of the program. It ensures that all objects reachable from an object behind the membrane are also behind the membrane. Figure 3.2 shows a membrane (dashed line) around `target` implemented by the wrapper `proxy`. Each target object is represented by a wrapper object (dotted line) outside the membrane and each property access through a wrapper (e.g., `proxyA.x`) returns a wrapper on demand. Therefore, after installing the membrane, no *new* direct references to target objects behind the membrane become available.

An *identity preserving membrane* is a membrane that furthermore guarantees that no target object has more than one proxy. Thus, proxy identity outside the membrane reflects target object identity inside. For example, if `targetA.x.z` and `targetA.y` refer to the same object, i.e. `targetA.x.z===targetA.y`, then also `proxyA.x.z` and `proxyA.y` refer to the same wrapper object with `proxyA.x.z===proxyA.y`.

Both kinds of membranes can be implemented with JavaScript proxies. Maps associate target objects with their corresponding proxy objects. This mechanism may be used to revoke all references to an object network at once [20, 81] or to enforce write protection on the objects behind the membrane [64].

3.3 Notes

The implementation of TreatJS relies on JavaScript proxies to enforce contracts on functions and objects. Function and object contracts are *delayed contracts* (cf. Section 4.3.1) as they must stay with the value until the value is used. In this case, the target value gets packed in a proxy along with the given contract and the proxy applies the contract in all contexts of use.

Using proxy objects is the common way to implement delayed contract monitoring. Proxies implement contracts in Racket's contract framework [45, Chapter 7], in Disney's JavaScript contract system *Contracts.js* [30], and in JSConTest2 for JavaScript [64].

I

Higher-Order Contracts for JavaScript

4 A TreatJS Primer

Use and meaning of contracts have a long tradition coinciding with different intuitions and different application scenarios of developers. We begin this part with a series of examples that explain how contracts are written and that illustrates the basic principles underlying design and implementation of our contract system.

TreatJS is a language-embedded, higher-order contract system for JavaScript [33], which enforces contacts by runtime monitoring. It supports all the standard features of existing contemporary contract systems in combination with a set of novel features and features that have not been implemented in this combination before. In particular, TreatJS provides the following core features:

Embedded contract-language. TreatJS is implemented as a library and deployed as a language extension. All aspects are written in JavaScript and accessible through a contract API. Hence, a programmer need not learn new syntax to state contracts.

Predicates are functions. Predicates are specified by plain JavaScript functions. TreatJS does not impose syntactic restrictions on predicates but expects them to terminate on all inputs. Predicates use the full expressive power of JavaScript, i.e., they can access the application state of a program (but they are not allowed to change it).

Contracts are projections. Contracts cannot interfere with the execution of a contract abiding host program. To guarantee not to exert side-effects on the host program, predicate code runs in a sandbox with fine-grained access control and with a configurable degree of isolation. Adding a contract to a program does either return the same outcome or it signals a contract violation.

Dynamic contract construction. Contract abstraction enables to construct and compose contracts at runtime. A contract abstraction is a JavaScript function that comprises a contract definition. Values can be passed to the abstraction and the abstraction returns a contract. Contract abstraction is particularly important to build recursive contracts, parameterized contracts, or stateful contracts.

Full interposition. Contracts are enforced uniformly in all contexts of use. TreatJS contracts guarantee full interposition for the full JavaScript language [33] and for all code regardless of its origin, including dynamically loaded scripts and code injected via `eval`. No source-code transformation or change in the JavaScript runtime system is required.

A small core API provides a set of essential core contracts in isolation. Core contracts avoid JavaScript specialties where possible. A convenience API on top of the core API enables to state more complex and convenient contracts. All contracts are first-class values that can be stored and further composed. They are dormant until they are asserted to a value.

4.1 Base Contracts

The *base contract* (aka flat contract) is the fundamental building block of all other contracts. It is built from a predicate on a single argument and asserting it to a value applies the predicate to that value.

```
let typeBoolean = Contract.Base(function(subject) {
  return ((typeof subject) === "boolean");
},"typeBoolean");
let typeString = Contract.Base(function(subject) {
  return ((typeof subject) === "string");
},"typeString");
let typeFunction = Contract.Base(function(subject) {
  return ((typeof subject) === "function");
},"typeFunction");
let Any = Contract.Base(function(subject) {
  return true;
},"Any");
```

Listing 4.4 Some utility contracts.

In TreatJS, predicates are defined by plain JavaScript functions because any return value has an inherent boolean interpretation generally known as *truthy* or *falsy*[1]. A predicate holds on a value if applying the function to the value evaluates to a truthy value.

For example, the following function checks if its argument is of type number.

```
function isNumber (subject) {
  return ((typeof subject) === "number");
}
```

Listing 4.1 Example of a predicate function.

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

```
let typeNumber = Contract.Base(isNumber, "typeNumber");
```

Listing 4.2 Construction of a base contract.

The second argument passed to the constructor is an optional name for later user in error messages. Here, `Contract` is a wrapper object that encapsulates all constructors for TreatJS contracts. It also contains an `assert` function that attaches a contract to a *subject value*. Assigning a base contract applies the predicate to that value.

We call a base contract *immediate* because it evaluates immediately when asserted to a value. Customarily, `assert` returns the original value until it observes a contract violation. A failing predicate signals a violation blaming the subject value for not fulfilling the predicate. A base contract will never blame its *context*, which is the code that uses the subject.

In the following example, we first assert `typeNumber` to a number which returns the number value itself, whereas the second assertion throws a contract violation blaming the string value `"a"` for violating the `typeNumber` contract.

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

Listing 4.3 Blame assignment of a base contract.

Listing 4.4 defines some base contracts for later use. Like `typeNumber`, the contract `typeBoolean`, `typeString`, and `typeFunction` check their argument to a boolean, string or function value, respectively. The `Any` contract accepts any value.

[1] The following values are *falsy*: `false`, `0` (zero), `""` (empty string), `undefined`, and `NaN`. All other values *truthy*.

4.2 Contract Constructors

The *contract constructor* is the second fundamental building block of our contract system. It is for building *contract abstractions* which are used to build stateful contracts, dependent contracts, parameterized contracts, and recursive contracts. The contract constructor is not a contract per se, but when applied to some values it reduces to a contract.

Contract constructors are particularly important because, by default, base contracts (predicates) cannot access values define outside it own scope, except the subject value. Contract constructors make eternal values available inside of predicate code (cf. Section 4.5).

Like a base contract, it takes a JavaScript function and an optional name as an argument. Unlike a base contract, the function maps zero or more parameters to a contract. The function body may contain contract definitions and it has to return a contract. Each contract defined inside shares the local variables and parameters visible in the function's scope. The second argument passed to the constructor is an optional name for later user in error messages.

For example, the contract constructor `TypeOf` builds a base contract from a specific type.

```
let TypeOf = Contract.Constructor(function(type) {
  return Contract.Base(function(value) {
    return ((typeof value) === type);
  }, `TypeOf ${type}`);
}, "TypeOf");
```

Listing 4.5 Construction of a contract constructor.

To obtain a contract from a constructor we apply the constructor to argument values. The constructor can be used as a normal JavaScript function. For example, to create a contract that checks for number values we apply the constructor to the type name `"number"`.

```
let typeNumber2 = TypeOf("number");
```

Listing 4.6 Obtain a contract from a contract constructor.

This constructor provides an alternative construction of the `type`-contracts in Listing 4.4. Instead of writing similar base contracts, the parameterized constructor can be used to construct different versions of the same base contract.

Apart from building parameterized constructors, contract constructors are particularly important to make values available inside of a predicate. By default, predicates are not able to access values defined outside of the predicates scope. This restriction prohibits the formation of some useful contracts.

For example, the `instanceOfArray` contract needs to access the global `Array` object to check if its argument is an instance of `Array`. Without giving a concrete permission, `instanceOfArray` would not be able to access the `Array` object.

```
let InstanceOf = Contract.Constructor(function(constructor) {
  return Contract.Base(function(object) {
    return (object instanceof constructor);
  }, `InstanceOf ${constructor.name}`);
}, "InstanceOf");
let instanceOfArray = InstanceOf(Array);
```

Listing 4.7 Definition of the `InstanceOf` constructor.

Listing 4.8 shows some utility constructors for later use. Constructor `GreaterThan` returns a base contract that checks if a value is greater than a specific target value. Constructor `Between` returns a base contract that checks if a value is in a certain range, and constructor `Length` returns a base contract that checks the `length` property of an object.

```
let GreaterThan = Contract.Constructor(function(value) {
  return Contract.Base(function(subject) {
    return (subject < value);
  }, `GreaterThan ${value}`);
}, "GreaterThan");
let Between = Contract.Constructor(function(min, max) {
  return Contract.Base(function(subject) {
    return (min <= subject) && (subject <= max);
  }, `Between ${min} and ${max}`);
}, "Between");
let Length = Contract.Constructor(function(length) {
  return Contract.Base(function(subject) {
    return (subject.length === length);
  }, `Length ${length}`)
}, "Length");
```

Listing 4.8 Some utility constructors.

```
let Even = Contract.Constructor(function(Math) {
  return Contract.Base(function(subject) {
    return (Math.abs(subject) % 2 === 0);
  }, `Even`);
}, "Even")(Math);
let Odd = Contract.Constructor(function(Math) {
  return Contract.Base(function(subject) {
    return (Math.abs(subject) % 2 === 1);
  }, `Odd`);
}, "Odd")(Math);
```

Listing 4.9 Some utility contracts (cont'd).

Furthermore, Listing 4.9 shows some utility contracts built from contract constructors. Contract `Even` checks for even numbers and contract `Odd` checks for odd numbers, respectively. Both base contracts use the `abs` function from the global `Math` object to obtain the absolute value of a given subject. The constructor makes `Math` available inside of the predicates.

4.3 Higher-Order Contracts

The base contracts in Section 4.1 and Section 4.2 specify flat statements on primitive values that may be true or false depending on the subject values. Even though straightforward assertions of base contracts may also test properties on functions and objects, they are not expressive enough to state higher-order properties of functions and objects that cannot be checked immediately. For example, a contract should be able to express that a function `plus` maps two number values to a number value or that the access to the `length` property of an array object always returns a number value. Hence, higher-order contracts are needed to address first-class functions and other advanced abstractions.

To this end, TreatJS provides several kinds of *higher-order contracts*. In general, a higher-order contract is a contract that

- takes one or more contracts for the domain of an operation and that
- returns a contract for the range of an operation.

4.3.1 Function Contracts

Following Findler and Felleisen's work on contracts for higher-order functions [41], a *function contract* is built from zero or more contracts for the domain of a function (one contract per argument) and one contract for the range (return) of a function.

We call a function contract *delayed* because asserting it to a value does not immediately signal a violation. Asserting the function contract amounts to asserting the domain contracts to the argument values and asserting the range contract to the return value of each call. A function contract applied to a non-function value will never signal a contract violation.

For example, consider the function `plus` which first checks if both arguments are greater than `0` and second, depending on the result of this test, it either applies JavaScript's native addition operator `+` to its arguments or it returns a string message.

```
1 function plus(x, y) {
2   return (x>0 && y>0) ? (x + y) : "Error";
3 }
```

Listing 4.10 Definition of function `plus`.

As a running example we develop several contracts for `plus`. Our first contract restricts input and output to values of type `"number"`.

```
1 Contract.Function([typeNumber, typeNumber], typeNumber);
```

Listing 4.11 Construction of a function contract.

`Contract.Function` is a constructor for function contracts. Its first argument is an array that maps an argument index (starting from zero) to a contract, whereas the second argument is a contract for the return of a function.

To assert a function contract, `Contract.assert` wraps the subject value (in this case function `plus`) in a special *contract proxy* that mediates all uses of the wrapped function. For delayed contracts, `Contract.assert` does not return the subject value itself, but it returns a value that behaves identical to the subject value, until it observes a contract violation.

```
let plusNumber = Contract.assert(function plus(x, y) {
  return (x>0 && y>0) ? (x + y) : "Error";
}, Contract.Function([typeNumber, typeNumber], typeNumber));
```

Listing 4.12 Assertion of a function contract.

So, `plusNumber` accepts any argument that satisfies `typeNumber` and promises to return a value that satisfies `typeNumber`. Calling `plusNumber` with an argument that violates its contract, then the function contract throws a contract violation blaming the *context* (the caller of a function) for providing the wrong kind of argument. In case that all arguments are ok, but the return value does not satisfy its contract, then the function contract throws a contract violation blaming the *subject* (the function itself) for returning a wrong value. The following example demonstrates blame assignment of a function contract.

```
plusNumber(1,1); // accepted, returns 2
plusNumber(0,1); // violation, blame the subject
plusNumber("a","b"); // violation, blame the context
```

Listing 4.13 Blame assignment of a function contract.

This example also demonstrates another particularity of contracts. A contract deems to be satisfied until it is violated. Even though we know that `plus` does not satisfy its contract, no blame is allocated until `plus` is used with a number less than or equal to `0`.

Apart from having base contracts on the domain and range of a function, *higher-order function contracts* are also possible and can be defined in the same way. For example, a function `addOne`, which takes a `plus` function and a single number value as an argument, may be specified by the following contract.

```
let addOneNumber = Contract.assert(function addOne(plus, z) {
  return plus(z, 1);
}, Contract.Function([Contract.Function([typeNumber, typeNumber], typeNumber)),
    typeNumber], typeNumber));
```

Listing 4.14 Nesting of function contracts.

Higher-order function contracts open up new ways to violate a contract. For example, function `addOne` may call `plus` with a non-number value.

In general, the context of a function (the caller) is responsible for calling the function with arguments that satisfy the domain specification of a function, and it has to use the function's return according to its specification. Likewise, the subject (the function) is responsible for using the argument values according to their specification and to return a value that satisfied the range specification.

```
addOneNumber(plus, 1); // accepted, returns 2
addOneNumber(plus, 0); // violation, blame the context
```

Listing 4.15 Blame assignment of a higher-order function contract.

The second call of `addOneNumber` throws a contract violation blaming the context of `addOneNumber`. This is because the context did not call `addOneNumber` with a `plus` function that satisfies `Contract.Function([typeNumber, typeNumber], typeNumber)`.

To demonstrate subject blame, we use a slightly different version of `addOne` which calls `plus` with its own argument `z` and the string value `"1"`.

```
let addOneBroken = Contract.assert(function addOne(plus, z) {
  return plus(z, "1");
```

```
}, Contract.Function([Contract.Function([typeNumber, typeNumber], typeNumber),
    typeNumber], typeNumber));
addOneBroken(plus, 1); // violation, blame the subject
```

Listing 4.16 Blame assignment of a higher-order function contract (cont'd).

Before concluding this section, it is important to mention that all examples demonstrate a convenient construction of a function contract. In JavaScript, functions do not have a fixed arity and arguments are passed to a function in an array-like `arguments` object. Thus, `Contract.Function` implicitly creates an object contract (cf. Section 4.3.2) with n contracts for the first n arguments from that array given as the first argument. Internally, the core function contract applies the object contract to the `arguments` array before it is passed to the function.

However, the core function contract accepts any contract on its domain. So, for example, another function contract to `plus` may check that the function is called with exactly two arguments, as the following example demonstrates.

```
let plusTwoArgs = Contract.assert(function plus(x, y) {
  return (x>0 && y>0) ? (x + y) : "Error";
}, Contract.Function(Length(2), Any));
```

Listing 4.17 Using the core function contract.

Here, the base contract from `Length(2)` is directly applied to the `arguments` array. Calling `plusTwoArgs` with more or less than two arguments will throw a contract violation blaming the context for providing the wrong number of arguments.

4.3.2 Object Contracts

An *object contract* is a simple key/contract map. It is defined by a mapping from key values (which are either property names, array indices, or symbols) to contracts.

Like a function contract, an object contract is delayed. All contracts in the mapping are dormant until the associated property is addressed by a property read or property write operation. When reading a property, the contract gets asserted to the accessed value before it is given to the context, whereas writing a property asserts the contract to the new value.

The following example demonstrates the construction of an object contract that indicates that the `length` property of an object has to be a value of type `"number"`. The constructor `Contract.Object` takes an enumerable JavaScript object (either an array or a normal object) that maps key values to contracts.

```
Contract.Object({length:typeNumber});
```

Listing 4.18 Construction of an object contract.

Analog to type systems, the object contract is satisfied by all objects that possess a field `length` containing a number value. Thus, it is safe to read `length`. However, there might be other, unspecified, fields beside `length`. Reading an unspecified property needs further assumptions about the underlying operational semantics of the programming language. In particular, there are two alternatives:

- Access to a non-existing property leads to a runtime error.
- Access to a non-existing property does not lead to a runtime error.

While traditional record calculi assume the first alternative, the latter one might be more sensible for JavaScript as reading an undefined property always returns `undefined` instead of

raising a runtime error. JavaScript developers frequently access undefined properties to test for non-existing fields. Furthermore, the existence of fields may change during execution.

TreatJS object contracts enable both possibilities. By default, an object contract is *weak*, i.e., it does not restrict properties that are not mentioned in the contract. However, a special boolean flag given to constructor creates a *strict* object contract that throws a contract violation blaming the context when accessing a non-existing property.

```
1 Contract.Object({length:typeNumber}, true);
```

Listing 4.19 Construction of a strict object contract.

For now, we focus only on weak object contracts. One can argue that the weak object contract is a special kind of a strict object contract that has an implicit `Any` contract in force for all properties not mentioned by the contract.

Blame assignment for object contracts is inspired by Reynold's approach on using built-in function interfaces to access reference cells [91]. Property reads and property writes are represented by internal getter and setter functions, both of which apply a function contract with the property's contract, but on different positions. The getter function of a property x has the form $x : \top \to \mathcal{C}$, whereas the setter function asserts $x : \mathcal{C} \to \top$. Here, $\to$ indicates a function contract, $\mathcal{C}$ is the property's contract, and $\top$ is a contract that accepts any value.

Reading a property that violates the associated contract throws a subject blame, blaming the object for returning a value that does not fulfill its specification. However, if the context assigns a value that does not fulfill its specification, then the context gets blamed because the property contract is on the domain of the function contract. Asserting an object contract to a non-object value will never signal a violation.

The following example demonstrates blame assignment for an object contract.

```
1 let array = Contract.assert({length:"1"}, Contract.Object({length:typeNumber});
2 array.length; // violation, blame the subject
3 array.length = "1"; // violation, blame the context
```

Listing 4.20 Blame assignment for an object contract.

Apart from base contracts, object contracts may also contain any other contract. For example, a function property might be contracted with a function contract.

However, this leads to a more complex blame assignment as the responsibility rests with the context that assigns a function property and with the context that uses the function property. To demonstrate this, the following object contract specifies that the `plus` method of the `Arithmetic` object has to satisfy a certain function contract.

```
1 let Arithmetic = Contract.assert({/* some object */}, Contract.Object({
2   plus:Contract.Function([typeNumber, typeNumber], typeNumber)
3 });
```

Listing 4.21 Construction of an object contract (cont'd).

Now, let's assign the `plus` function from Section 4.3.1 to the `Arithmetic` object.

```
1 Arithmetic.plus = function plus(x, y) {
2   return (x>0 && y>0) ? (x + y) : "Error";
3 }
```

Listing 4.22 Assigning a function property.

Assigning a function value to `plus` asserts the function contract to that value. However, a function contract is a delayed contract, so that it is dormant until the function is used in

an application. Now, when reading the function property, the object contract asserts the function contract one more time.

This double-assertion is required because the first contract is related to the context that writes the property, whereas the second contract is related to the context using the function. The following example demonstrates the blame assignment of such a method.

```
Arithmetic.plus("a", "a") // violation, blames the context
Arithmetic.plus(0, 0) // violation, blames the context
```

Listing 4.23 Blame assignment for an object contracts (cont'd).

In both cases, we blame the context for violating the object contract. The first one is because the current context calls the plus function with arguments values violating the contract on `plus`, whereas the second blame is because the context previously assigned a malicious plus function which did not satisfy the object contract.
As the last example, an object contract might also be the domain contract of a function contract, which is the proper way of writing a function contract. The following code snippet shows an equivalent way of writing a function contract for function `plus` from Section 4.3.1

```
let plusNumber = Contract.assert(plus, Contract.Function(Contract.Object([
    typeNumber, typeNumber]), typeBoolean));
```

Listing 4.24 Construction of a function contract using an object contract.

4.3.3 Dependent Contracts

A *dependent contract* is a special function contact where the range contract depends on the function arguments. In TreatJS, dependent contracts are created using contract constructors that are invoked with the caller's arguments. The constructor binds the arguments values and returns a contract for the range of that function.

The following example demonstrates the construction of a dependent contract that specifies that a function `sort` has to return an array of the same length as its input array. The constructor `Contract.Dependent` can either be called with a single JavaScript function or with a contract constructor. Calling the constructor with function internally converts the function to a contract constructor.

```
let sort = Contract.assert(function sort(input, compareFunction) {
  /* not specified in detail */
}, Contract.Dependent(function(input, compareFunction) {
  return Contract.Base(function(output) {
    return (input.length === output.length);
  });
});
```

Listing 4.25 Construction of a dependent contract.

Dependent contracts in TreatJS do not check the domain of a function, as done in other contract systems. However, TreatJS can intersect the dependent contract with another function contract that checks the domain. Section 4.4 demonstrates such a contract.

4.3.4 Method Contracts

The *method contract* is another special function contract. In JavaScript, function application usually takes two arguments: a `this` value that binds an object for method calls and an

array-like object specifying the arguments of that call. Thus, TreatJS's convenience API provides a special method contract that includes a contract specification for `this`.

For example, the `sort` function of the built-in `Array.prototype` object might be specified by a method contract which requires that `sort` must be called as a method of an object that is an instance of `Array`. The method contract further specifies that the method's argument must be a function and that the method has to return an array.

```
1 Array.prototype.sort = Contract.assert(function sort(compareFunction) {
2   /* not specified in detail */
3 }, Contract.Method(InstanceOf(Array), [InstanceOf(Function)], InstanceOf(Array));
```

Listing 4.26 Construction of a method contract.

Like a function contract, the method contract constructor implicitly converts all arrays that were given as an argument to an object contract. All other contracts apply directly to the corresponding values.

4.4 Combination of Contracts

Beyond base, function, object, dependent, and method contracts, TreatJS provides built-in constructors for intersection and union contracts. Both are again contracts that can be further composed or contained in another contract.

Intersection and union contracts are inspired by the corresponding operators in type theory [19]. A value has an intersection type $\tau_1 \cap \tau_2$ if it has both types τ_1 **and** τ_2. Consequently, the context of that value can choose to use it either as a value of type τ_1 **or** τ_2. Intersection types are particularly important to model overloading and multiple inheritances of values. The dual of intersection types, union types [7], model the domain of overloaded types. Contrary to intersection types, a value has a union type $\tau_1 \cup \tau_2$ if it has type τ_1 **or** τ_2, whereas the context must be able to deal with both types, τ_1 **and** τ_2.

4.4.1 Intersection Contracts

In type theory, if a value has two types, then we can assign it an intersection type [19]. TreatJS provides a corresponding constructor for intersection contracts.

For base contracts, the *intersection* nicely coincides with the conjunction of the predicates as the subject value needs to satisfy both contracts. For example, the intersection of the base contracts `Positive` and `Even` tests for positive and even values.

```
1 let PositiveEven = Contract.Intersection(Positive, Even);
2 Contract.assert(0, PositiveEven); // violation, blame the subject
3 Contract.assert(1, PositiveEven); // violation, blame the subject
4 Contract.assert(2, PositiveEven); // accepted
```

Listing 4.27 Construction of an intersection contract.

The constructor `Contract.Intersection` builds an intersection contract from its arguments. The intersection contract enables to specify independent properties in different base contracts (predicate) and to use intersection to combine various base contracts to a specific contract. Obviously, it is also possible to write an equivalent "fat" predicate[2] that checks for positive and even values.

[2] A *fat predicate* is a predicate that tests more than one property.

```
let PositiveEven = Contract.Constructor(function(Math) {
  return Contract.Base(function(subject) {
    return ((Math.abs(subject) % 2 === 0) && (subject > 0));
  }, ‘Positive and Even‘);
}, "Positive and Even")(Math);
```

Listing 4.28 Construction of a base contract with a fat predicate.

Intersections for base contracts are not really existing as they can be pushed into the predicates. However, separate contracts improve modularity and reusability of contracts and merging intersections did not work for intersections of function or object contracts.

To demonstrate intersections of function contracts, let's recap function `plusNumber` from Section 4.3.1. Its contract restricts the arguments to number values and promises to return a number. However, JavaScript's plus operator `+` is overloaded and works for strings and numbers. It either produces the sum of numeric operands or it concatenates strings. Thus, function `plus` works for strings, too. In TreatJS, we can assign it an intersection contract that enables both inputs.

```
let plus = Contract.assert( function plus(x, y) {
  return (x>0 && y>0) ? (x + y) : "Error";
}, Contract.Intersection(
  Contract.Function([typeNumber, typeNumber], typeNumber),
  Contract.Function([typeString, typeString], typeString)
));
```

Listing 4.29 Construction of an intersection contract (cont'd).

Use and meaning of an intersection contract nicely coincide with the meaning of an intersection type. The context can choose to use `plus` either with arguments that satisfy `typeNumber` or `typeString`, i.e., the arguments must satisfy the union (cf. Section 4.4.1) of the domain contract. The context of an intersection gets blamed if the arguments fail both domain contracts, `[typeNumber, typeNumber]` and `[typeString, typeString]`. Contrarily, the subject (the function `plus`) needs to deal with both inputs. Depending on its input it must either return a string or a number value. It gets blamed if it does not satisfy the range contract of each function contract whose domain contract is satisfied. The following example demonstrates the blame assignment.

```
plus(1, 1); // accepted
plus("a", "a") // accepted
plus(true, true); // violation, blame the context
plus(1, "a"); // violation, blame the context
plus(0, 0); // violation, blame the subject
```

Listing 4.30 Blame assignment of an intersection contract.

Next, let's take a closer look into intersection contracts. In the previous example, both domain contracts are disjoint: no value satisfies `typeNumber` and `typeString` at the same time. So, the subject must either return a number or a string, depending on its input.

However, the intersection indicates that the subject is able to deal with both contracts. This means, if an argument satisfies both domain contracts, then the subject must return a value that also satisfies both range contracts. The following example demonstrates this.

```
let addOne = Contract.assert(function addOne(x) {
  return (x+1);
}, Contract.Intersection(
  Contract.Function([Even], Even),
```

```
  Contract.Function([Positive], Positive)
));
plus(0); // violation, blame the subject
plus(1); // accepted
plus(-1); // violation, blame the context
plus(2); // violation, blame the subject
```

Listing 4.31 Blame assignment of an intersection contract (cont'd).

In TreatJS it is also possible to build the intersections of object contracts. However, the intersection of object contracts might be confusing without further clarification. In TreatJS, object contracts are either *weak* or *strict* (cf. Section 4.3.2), i.e., they either allow access to not specified properties, or they don't. Strict object contracts throw a contract violation when reading or writing to an unspecified property, whereas weak object contracts have an implicit `Any` contract for all undefined property in force.

To demonstrate this, let's consider the following object specification.

```
let object = Contract.assert({x:true, y:true}, Contract.Intersection(
  Contract.Object({x:typeNumber}),
  Contract.Object({y:typeNumber})
));
```

Listing 4.32 Intersection of two object contracts.

As before, the subject must satisfy both contracts, whereas the context can choose to satisfy either of them. However, the blame behavior seems to be confusing at first glance.

```
object.x // violation, blame the subject
object.x = true // accepted
```

Listing 4.33 Blame assignment of an intersection of weak object contracts.

Reading `object.x` throws a contract violation blaming the subject because the subject must satisfy both contracts; this is `typeNumber` from the left contract and `Any` from the right contract. In this example, there is no difference between weak and strict object contracts when reading a property. The subject needs to satisfy all property contracts if there are any.

Contrary to our expectations, writing `object.x = true` does not throw a contract violation. This is because the context can choose either to satisfy `typeNumber` from the left or `Any` from the right contract. However, a similar intersection with strict object contracts would report a contract violation blaming the context because of both, `Contract.Object({x:typeNumber})` and `Contract.Object({y:typeBoolean})`, report a context failure.

In this example, the intersection of two strict object contracts is equivalent to a single object contract `Contract.Object({x:typeNumber, y:typeBoolean})` which specifies both properties. However, the intersection enables to have more than one contract per property, which is not possible with a single object contract.

```
let object = Contract.assert({x:1, y:1}, Contract.Intersection(
  Contract.Object({x:typeNumber}),
  Contract.Object({x:Positive})
));
```

Listing 4.34 Intersection of two object contracts (cont'd).

Furthermore, consider the intersection of two object contracts that specify a method.

```
let object = Contract.assert({/* some object */}, Contract.Intersection(
  Contract.Object({f:Contract.Function([typeNumber], typeNumber)}),
```

```
  Contract.Object({g:Contract.Function([typeBoolean], typeBoolean)})
));
```

Listing 4.35 Intersection of two object contracts (cont'd).

When evaluating `object.f(1)` we get a subject failure if `object.f` does not return something of type number. But, when evaluating `object.f(true)`, then we get no blame because there is no contract violation from `Contract.Object({g:Contract.Function([typeBoolean], typeBoolean)})`. Unlike the weak object contract, a similar intersection of strict object contracts would throw a contract violation blaming the context.

Apart from intersecting contracts of the same type, TreatJS also enables arbitrary combinations of contract. For example the intersection of a flat contract with a function contract. One prominent use of this is the construction of a *real* function contract that can only be applied to function values.

```
let RealFunction = Contract.Constructor(function(functionContract) {
  return Contract.Intersection(typeFunction, functionContract);
}, functionContract.toString());
```

Listing 4.36 Construction of a real function contract.

The left-hand side of the intersection is a base contract that checks if its subject value is a function. A base contract is an immediate contract that is checked right away when asserted to a value, whereas the function contract remains on the function until the function is used. However, if the subject value is not a function, then the base contract immediately reports a subject violation, which in turn leads to a subject violation of the whole intersection.

At this point, we have to mention that, even if TreatJS enables unrestricted combinations of contract, it is not possible to combine contracts arbitrarily. TreatJS requires its core contracts to be in a *canonical form* to achieve the correct blame behavior and to improve the efficiency of contract monitoring. Thus, it restricts intersection contracts to intersections of an immediate contact and any other contract or to an intersection of delayed contracts[3]. However, TreatJS provides a special constructor function `Contract.Intersection.from` that allows building intersections of arbitrary contracts. `Contract.Intersection.from` *normalizes* contracts and returns a canonical intersection contract equivalent to the given intersection.

Normalization pulls unions out of intersections and separates immediate contracts from the delayed contracts of an intersection, such that the immediate contracts can be evaluated right away when asserted to a value, whereas the delayed contracts remain on the subject value until the subject is used. Section 5.4.3 provides more details on this.

4.4.2 Union Contracts

Union contracts, the dual of intersection contracts, are also used for connections between contracts. Union contracts model the domain of overloaded values, and in some situations, there is an equivalence that enables to rephrase an intersection contract with a union contract or vice versa. Again, there is a connection to union types.

For example, consider the intersection contract on a compare function that either compares strings or numbers and that promises to always returns a boolean.

```
let compare = Contract.assert(function compare(x, y) {
```

[3] The convenience API also provides an intersection contract of immediate contracts, but this contract is not necessary and only available to improve usability.

```
2   return (x > y);
3 }, Contract.Intersection(
4   Contract.Function([typeNumber, typeNumber], typeBoolean),
5   Contract.Function([typeString, typeString], typeBoolean)
6 ));
```

Listing 4.37 Specification of function `compare`.

Exploiting the well-known type equivalence $(\tau_1 \to \tau_3) \cap (\tau_2 \to \tau_3) \equiv (\tau_1 \cup \tau_2) \to \tau_3$ [7], an equivalent contract can be written as follows.

```
1 let compare = Contract.assert(function compare(x, y) {
2   return (x > y);
3 }, Contract.Function(
4   Contract.Union(
5     Contract.Object([typeNumber, typeNumber]),
6     Contract.Object([typeString, typeString])
7   ), typeBoolean
8 ));
```

Listing 4.38 Alternative specification of function `compare`.

Here, `Contract.Union` builds the union of two contracts. As before, TreatJS enables to build the union of arbitrary contract and does not restrict the union to contracts of the same type. Furthermore, all top-level union contracts need not be normalized.

Just like intersections, the *union* of two base contracts can be pulled down to a disjunction on the enclosed predicates. The subject can choose to satisfy either of them.

```
1 let PositiveOrOdd = Contract.Union(Positive, Odd);
2 Contract.assert(0, PositiveOrOdd); // violation, blame the subject
3 Contract.assert(1, PositiveOrOdd); // accepted
4 Contract.assert(2, PositiveOrOdd); // accepted
```

Listing 4.39 Union on base contracts.

But union contracts are also applicable to function and object contracts. A function satisfies the union of two function contracts if it satisfies either of them, whereas the context of a union must deal with both contracts. The following example demonstrates its blame behavior.

```
1 let mod3 = Contract.assert(function mod3(x) {
2   return (x % 3);
3 }, Contract.Union(
4   Contract.Function([Even], Even),
5   Contract.Function([Positive], Positive)
6 ));
7 mod3(4); // accepted, returns 1
8 mod3(1); // violation, blame the context
9 mod3(6); // violation, blame the subject
```

Listing 4.40 Blame assignment of a union contract.

In line 7 the context calls `mod3` with a positive and even number and the subject returns a positive odd number. The contract is satisfied as the context satisfies both contracts and the subject can choose to fulfill either `Even` **or** `Positive`.

In the second call the context gets blamed because it fails to satisfy both domain contracts, `Even` **and** `Positive`.

In the last use, the context calls `mod3` with a positive and even number, but the subject returns `0`, an even but not a positive number. Even though the subject of a union can choose

to fulfill either of both contracts, it is not allowed "flip" between both specifications. A misbehaving function gets blamed on its first alternation. Here, `mod3` decides to satisfy the `Contract.Function([Positive], Positive)` contract. As in the last example, it attempts to return an even, but not a positive number, TreatJS throws a contract violation, blaming the subject for return an inappropriate value.

Finally, let's look at the union of two object contracts. As an example we again consider object `{x:true, y:true}` with the following specification.

```
let object = Contract.assert({x:true, y:true}, Contract.Union(
  Contract.Object({x:typeNumber}),
  Contract.Object({y:typeBoolean})
));
```

Listing 4.41 Union of two object contracts.

To recap, this is a weak object contract, which allows reading of unspecified properties.

Like before, the subject can choose to satisfy either of both contracts (but it is not allowed to alternate between both contracts), whereas the context must deal with both sides of the union. So, reading `object.x` does not throw a contract violation. The contract is satisfied as the right side of the union does not restrict property `x`. However, if the subject decides to satisfy the right side, subsequent read access to property `y` throws a contract violation blaming the subject. The following code snippet demonstrates this behavior.

```
object.x; // accepted, returns true
object.y; // violation, blame the subject
```

Listing 4.42 Blame assignment of a union contract.

Furthermore, writing a property requires that all property contracts are satisfied.

```
object.x = true; // violation, blame the context
```

Listing 4.43 Blame assignment of an union contract (cont'd).

Writing `object.x` throws a contract violation as the context did not assign a number value.

For a strict object contract, every property read and property write would immediately result in a contract violation. This is because both contracts specify different property names, and reading an unspecified property immediately reports a context failure. As the context of a union always has to satisfy both contracts, the union of two strict object contracts requires that all readable and writeable properties are specified in both contracts.

Reading and writing a function property results in an equivalent behavior.

4.5 Sandboxing Contracts

In most existing contract systems a contract remains invisible until the contract monitor observes a violation. This also includes that contracts and predicates do not influence the normal program execution or write values that are also visible to the host application.

In TreatJS, predicates are specified by plain JavaScript functions, i.e., predicates use the full expressive power of JavaScript and there are no syntactic restrictions on predicates. However, the execution of a contract abiding host program should not be influenced by the evaluation of a terminating predicate inside of a base contract. But, without any restrictions predicate code might manipulate values outside the predicate's scope or it might call a side-effecting function.

To illustrate this issue we rephrase the `typeNumber` contract from Section 4.1.

```
1 let maliciousTypeNumber = Contract.Base(function(subject) {
2   type = (typeof subject);
3   return (type === "number");
4 },"typeNumber");
```

Listing 4.44 Malicious implementation of `typeNumber`.

Here, a programmer first determines the type of the subject value and stores it in a variable before it compares the type names. But, the programmer missed adding the `let` keyword in front of the assignment in line 2. As the variable `type` might be defined in an enclosing scope, the assertion of `maliciousTypeNumber` could overwrite a variable defined outside its own scope. Thus, it might influence the normal program execution.

To prevent such unintended interference, TreatJS evaluates predicates in a *sandbox* with a configurable degree of isolation. By default, the sandbox reopens the predicate's closure and removes all external binding of variables. It encapsulates the predicate in a membrane that stops the evaluation if a predicate attempts to access variables defined outside the predicate's scope.

To demonstrate, the evaluation of `maliciousTypeNumber` stops and throws a sandbox violation as soon as the predicate attempts to write `type`.

```
1 Contract.assert(1, maliciousTypeNumber); // violation, access forbidden
```

Listing 4.45 Causing a sandbox violation.

However, read-only access is safe, and many useful contracts require access to objects or functions defined outside. For example, the `instanceOfArray` contract from Section 4.2 requires access to the global `Array` object to check if its subject is an instance of `Array`. Without giving a concrete permission, the sandbox rejects any access to variables defined outside the predicate's scope.

To grant permission, TreatJS provides the contract constructors (cf. Section 4.2). The contract constructor builds a contract abstraction. It consists of a function that maps a number of parameters to a contract. As a predicate, this function is evaluated in the sandbox. However, every contract (predicate) defined inside the constructor function shares the local variables visible in the constructor's scope. As every predicate defined inside of a constructor function is guaranteed to be without ties to the outside world, no further sandbox is required. Each value passed into the sandbox, either as a constructor argument or as a subject value, is wrapped in an identity-preserving membrane (cf. Section 3.2) to mediate access to the entire object structure.

In general, TreatJS provides three different *safety levels*. The most liberal level, *none*, deactivates the sandbox and enables any interference of predicate code. The *pure level* evaluates predicate code and constructor functions in a sandbox and wraps every value in a membrane that prohibits write access but allows unrestricted read access, whereas the *strict level* prevents any kind of access. Unfortunately, any read access to a JavaScript object might be the call to a side-effecting getter function or the call of a side-effecting proxy trap. While getter functions can be recognized, nested proxy traps always remain undetected. Thus, only the strict level guarantees full noninterference as it restricts any access to the target object, but it also prohibits some useful operations, for example, `instanceof` tests[4].

[4] In JavaScript, the `instanceof` test calls the `[Symbol.hasInstance]` method of a constructor function to determines if a constructor recognizes an object as its instance. For example, `a instanceof A` is equivalent to `A[Symbol.hasInstance](a)`.

The pure level, in contrast, enables read access but places several restrictions on it. First, any return of a read must be wrapped in the membrane. Second, every function (this also includes function properties and getter functions) must be evaluated in our sandbox. To this end, the membrane mediates any function application of wrapped objects and evaluates the function in our sandbox.

4.6 Lax, Picky, and Indy Semantics

TreatJS distinguishes three different Monitoring semantics: *Lax*, *Picky*, and *Indy*. The general idea of these semantics are drawn from the literature [41, 10, 26], which introduces different kinds of evaluation semantics to handle correctness and completeness of higher-order dependent contracts. TreatJS provides a generalization of those semantics.

One ground rule of contract monitoring is that a contract abiding host program should not be influenced by the introduction of contracts. But, what happens if a contract violates another contract? Following this ground rule, contract violations in other contracts should be ignored as they are not part of the host program. On the other hand, this makes it possible to violate a contract without consequences: you only need to put the malicious execution in a predicate of a base contract and to assert this base contract to some value.

In TreatJS, predicate evaluation takes place in a sandbox that prohibits side-effects on the host program (which also includes thrown exceptions). However, this might enable that a function is used against its specification without recognizing the violation. To overcome this, different blame semantics define the blame behavior of contract violations in contracts.

To make this discussion concrete, consider the definition of function `id`.

```
1 let id = Contract.assert(function id(x) {
2   return x
3 }, Contract.Function([typeNumber], typeNumber));
```

Listing 4.46 Definition of function `id`.

Next, let's consider the definition of a base contract `idTest` which tests function `id`.

```
1 let idTest = Contract.Base(function test(id) {
2   return id("a") === "a";
3 }, "ID Test");
```

Listing 4.47 Definition of base contract `idTest`.

The *Lax* semantics erases all contract monitors on values that pass the sandbox membrane. This is correct because it guarantees that a well-behaved program never gets blamed for a violation taking place in a predicate, but it is not complete as it swallows contract violations in predicates and constructors. The following example demonstrates this behavior.

```
1 Contract.assert(id, idTest); // accepted
```

Listing 4.48 Blame assignment of *Lax* semantics.

The *Picky* semantics, in contrast, is complete but not correct. It preserves all contract monitors on values, and it reports every violation, but it might wrongly blame the program for violations that happen in predicate code. The following example demonstrates blame behavior with picky semantics.

```
1 Contract.assert(id, idTest); // violation, blame the context
```

Listing 4.49 Blame assignment of *Picky* semantics.

The *Indy* semantics, an extension of *Picky*, introduces a third player (the contract itself) in addition to the existing players context and subject. Passing a value into the sandbox reorganizes the monitor such that wrong uses of that value blame the ill-behaved contract, whereas subject failures still blame the subject. The Indy semantics is correct and complete. The following snippet demonstrates its outcome.

```
Contract.assert(id, idTest); // violation, blame the contract "ID Test"
```

Listing 4.50 Blame assignment of *Indy* semantics.

Furthermore, subject violations are also possible, as the following example demonstrates.

```
let addOne = Contract.assert(Contract.assert(function addOne(plus, z) {
  return plus(z, 1);
}, Contract.Dependent(function(plus, z) {
  return Contract.Base(function(subject) {
    return plus(z, "a")  === z + "a";
  },  "Plus Test");
})), Contract.Function([Contract.Function([typeNumber, typeNumber], typeNumber),
    typeNumber], typeNumber)));
```

Listing 4.51 Assertion of a function and a dependent contract.

Function `addOne` takes a `plus` function and a number value and applies function `plus` to its second argument and the number value `1`. The specification of `addOne` takes two contracts. First, a dependent contract which creates a base contract for the return of function `addOne`, and second a function contract that specifies domain and range of `addOne`.

When calling `addOne` with a `plus` function and a number value, the outermost contract first applies `Contract.Function([typeNumber], typeNumber)` and `typeNumber` to `addOne`'s arguments, before it passes the arguments to the dependent contract. As the contract on `plus` is a delayed contract, it remains on the function when passing the function to the dependent contract. More details on the evaluation order of contracts are given in Section 5.4.5.

It, therefore, follows that the base contract which is returned from the dependent contract violates the contract `plus` by calling `plus` with values that do not satisfy the `typeNumber` contract. However, the result of this violation depends again on the chosen blame semantics, as the following code snippets demonstrate.

```
addOne(function plus(x, y) {
  return (x + y);
}, 2) // accepted, returns 3
```

Listing 4.52 Blame assignment of *Lax* semantics (cont'd).

As before, the Lax semantics removes all contract monitors and does not report a contract violation.

```
addOne(function plus(x, y) {
  return (x + y);
}, 2) // violation, blame the subject
```

Listing 4.53 Blame assignment of *Picky* semantics (cont'd).

In sharp contrast to Lax, the Picky semantics reports a contract violation blaming the subject value for using its arguments against its specification. Even though it is evident that is it not the subject's fault, the subject gets wrongly blamed for misconducts of its own contract.

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

```
}, 2) // violation, blame the contract "Plus Test"
```

Listing 4.54 Blame assignment of *Indy* semantics (cont'd).

Fortunately, the Indy semantics blames the base contract `"Plus Test"` for violating the contract on `plus`. However, not every contract violation inside of a contract is in the responsibility of the contract. Context and subject blames are still possible. For example, when calling `addOne` with a misbehaving `plus` function.

```
let addOne = Contract.assert(function addOne(plus, z) {
  return plus(z, 1);
}, Contract.Intersection(
    Contract.Dependent(function(plus, z) {
      return Contract.Base(function(subject) {
        return subject == plus(z, 1);
      },  "Plus Test");
    }),
    Contract.Function([Contract.Function([typeNumber], typeNumber), typeNumber],
    typeNumber)
  )
);
addOne(function plus(x, y) {
  return "1";
}, 2) // violation, blame the context
```

Listing 4.55 Blame assignment of *Indy* semantics (cont'd).

Calling `addOne` with a faulty `plus` function that always returns a string value reports a contract violation blaming the context as usual.

TreatJS uses Indy semantics by default, but any other blame semantics can be selected when initializing TreatJS. More details about the blame semantics are given Section 5.5.

4.7 Compatibility of Contracts

Apart from different monitoring semantics (cf. Section 4.6) a *compatibility test* for contracts on values that were passed to another contract (predicate) is needed. To demonstrate the need for a compatibility test, let's consider the intersection of a function contract and a dependent contract, as shown in the following example.

```
let addOne = Contract.assert(function addOne(plus, z) {
  return plus(z, 1);
}, Contract.Intersection(
    Contract.Dependent(function(plus, z) {
      return Contract.Base(function(subject) {
        return plus(z, "1");
      },  "Plus Test");
    }),
    Contract.Function([Contract.Function([typeNumber], typeNumber), typeNumber],
    typeNumber)
  )
);
```

Listing 4.56 Intersection of a function and a dependent contract.

As before, function `addOne` takes a `plus` function and a number value and applies function `plus` to its second argument and the number value `1`. Its specification is the intersection of a dependent contract and a function contract.

Viewed individually, `addOne` satisfies both contracts: the function contract and the dependent contract. However, without any compatibility test, it will not satisfy the intersection contract of both. This is because TreatJS sequentially unrolls the intersection contract from left to right and when calling `addOne` it first applies the domain contract of the outermost contract, which is the function contract in this case.

As the contract on `plus` is a delayed contract, it remains on the function value when the value is passed to the dependent contract. As the contained predicate calls plus with a non-number value, it will obviously violate the function contract on `plus`. However, this function contract belongs to the other side of an intersection and should therefore not be in force when evaluating the dependent contract.

Thus, TreatJS applies a compatibility test whenever a contracted value is used in another contract. This compatibility test drops all contracts that semantically belong to a parallel intersection or union of the same top-level assertion. It will not drop contracts of the same side of an intersection or union and it will not remove contracts that arise another top-level assertion. So, calling `addOne` will not throw a contract violation.

```
addOne(function plus(x, y) {
  return (x+y);
}, 1); // accepted, return 1
```

Listing 4.57 Intersection of a function and a dependent contract (cont'd).

Obviously, when using the Lax monitoring semantics, a compatibility test is not required as Lax always removes all contract monitors on values passed to another contract. However, Picky and Indy require this compatibility test to make intersection and union symmetric and to avoid a mutual influence of contracts in an intersection or union. More technical details about the compatibility test are given in Section 5.6.

4.8 Convenience Contracts

In addition to the core contracts previously mentioned in this chapter, TreatJS provides a *Convenience API* with contracts that build on TreatJS's core contracts.

4.8.1 Invariant Contracts

Formally, an *invariant* is a condition which might be true or false during the execution of a program. From a contract's perspective, an invariant is some kind of delayed predicate that is checked on every use of a contracted subject value. Invariants are especially useful to enforce a particular property on an object during program execution.

To demonstrate this, let's consider the definition of a binary tree `root`, as defined by `Node` and `Leaf` elements in Listing 4.58. Each `Node` element consists of a value field, a left child, and a right child, whereas a `Leaf` element only consists of a value field. Their `height` property returns the height of the element in the binary tree.

```
let root = new Node(2, new Node(1, new Leaf(0), new Leaf(0)), new Leaf(0));
```

Listing 4.59 Definition of a binary tree.

Furthermore, consider the definition of base contract `isBalanced` which checks if a given node is balanced.

```
let isBalanced = (Contract.Constructor(function(Node, Math) {
  return Contract.Base(function isBalanced(node) {
    if(node instanceof Node) {
```

```
function Node (value, left, right) {
  this.value = value;
  this.left  = left;
  this.right = right;
}
Node.prototype = {
  get height() {
    return (Math.max(this.left.height,this.right.height)+1);
  }
}
function Leaf(value) {
  this.value = value;
}
Leaf.prototype = {
  get height() {
    return 0;
  }
}
```

Listing 4.58 Implementation of a `Node` and `Leaf` element.

```
      const {value, left, right} = node;
      const lhs = left.height;
      const rhs = right.height;
      return isBalanced(left) && isBalanced(right) && (Math.abs(lhs - rhs) <= 1);
    } else {
      return true;
    }
  } ,"isBalanced");
}, "isBalanced"))(Node, Math);
```

Listing 4.60 Definition of the `isBalanced` contract.

The predicate recursively checks if the difference between the height of the left and the height of the right node is smaller or equals to one. To check if the `root` is a balanced we simply assert `isBalanced` to `root`, as the following example demonstrates.

```
let balanced_root = Contract.assert(root, isBalanced); // accepted, returns root
```

Listing 4.61 Assertion of the `isBalanced` contract.

Unfortunately, this check only applies once and all subsequent property updates to `root` might unbalance the binary tree, as the following example demonstrates.

```
balanced_root.left = new Node(3, new Node(2, new Node(1, new Leaf(0), new Leaf(0)),
     new Leaf(0)), new Leaf(0));
```

Listing 4.62 Unbalance a binary tree.

To overcome this, TreatJS enables to write an *Invariant Contract*, which is very similar to a delayed base contract that checks the predicate after each operation on the contracted subject value.

```
let BalancedNode = Contract.Invariant(isBalanced);
```

Listing 4.63 Definition of an invariant contract.

Now, we can assert the `BalancedNode` invariant on `root`.

```
let balanced_root = Contract.assert(root, BalancedNode);
```

Listing 4.64 Assertion of an invariant contract.

The invariant is checked on every use of `balanced_root`. So, it throws a context violation if we attempt to unbalance the binary tree like this:

```
let balanced_root.left = new Node(3, new Node(2, new Node(1, new Leaf(0), new Leaf
    (0)), Leaf(0)), Leaf(0)); // violation, blame the context
```

Listing 4.65 Blame assignment of an invariant contract.

The invariant contract blames the context for violating the predicate during the program execution. In general, invariant contracts will always blame the context for wrong uses. However, there is an initial predicate check in front that checks the adherence of the invariant at assertion time. This check will blame the subject value if it does not satisfy the invariant.

4.8.2 Recursive Contracts

A special feature of contract constructors is the possibility to build *recursive contracts*. In general, a recursive contract is a contract that is defined in terms of itself.

To motivate the need for recursion in contracts, let's recap the binary tree example from Subsection 4.8.1. The `BalancedNode` invariant enforces the `isBalanced` predicate on `root`. However, the invariant is only enforced on `root` directly, not on its children. So, it is still possible to unbalance the binary tree by modifying one of `root`'s child nodes.

To overcome this issue, TreatJS enables to define a recursive contract that recursively enforces the invariant on all child nodes. A recursive contract is build from a contract constructor that is later invoked with the recursive contract and which makes the contact available in another contract definition. Recursive contracts are closely related to recursive data types. The following example demonstrates the notation of a recursive contract.

```
let BalancedTree = (Contract.Constructor(function(BalancedNode) {
  return Contract.Recursive(function mu(self) {
    return Contract.And.from(BalancedNode, Contract.Object({
      value: typeNumber,
      left:  self,
      right: self
    }));
  });
}, "BalancedTree"))(BalancedNode);
```

Listing 4.66 Definition of a recursive contract.

The outermost contract constructor is only to make `BalancedNode` available inside of the sandbox. It returns the recursive contract, which is built from `Contract.Recursive`.

`Contract.Recursive` consumes another contract constructor[5], which is later invoked with the recursive contract itself. The given JavaScript function later returns a conjunction of the `BalancedNode` invariant and an object contract with the recursive contract in place. The object contract only implements the recursive walk-through, whereas the invariant checks the property. The `And` contract simply conjuncts two contracts.

[5] Here, `Contract.Recursive` implicitly converted the JavaScript function to contract constructor.

```
let balanced_tree = Contract.assert(root, BalancedTree);
```

Listing 4.67 Assertion of a recursive contract.

Now, the invariant is enforced on all node elements. Whenever we access a node starting from `balanced_tree`, the returned value is wrapped within the same contract as `balanced_tree`. So, subsequent read or write operation to one of its children will check the invariant as well.

```
let balanced_tree.right.left = new Node(3, new Node(2, new Node(1, new Leaf(0), new
    Leaf(0)), Leaf(0)), Leaf(0)); // violation, blame the context
```

Listing 4.68 Blame assignment of a recursive contract.

5 Contracts and Contract Monitoring

This chapter explains how contract monitoring in TreatJS works and how TreatJS determines the correct blame for a contract by the outcome of its constituents.

To this end, it defines $\lambda_{\mathcal{CON}}$, an untyped call-by-value lambda calculus with contracts that serves as a core calculus for contract monitoring. It first introduces the base calculus $\lambda_{\mathcal{J}}$, and then it proceeds to describe contracts and their semantics for the base calculus. Finally, it gives the semantics of contract assertion and blame propagation.

5.1 The Base Language $\lambda_{\mathcal{J}}$

This section introduces $\lambda_{\mathcal{J}}$, an untyped call-by-value lambda calculus with objects and object-proxies that serves as a core calculus for JavaScript. It defines its syntax and describes its semantics formally. The calculus is inspired by core calculi from the literature [68, 64, 66, 50, 54].

Figure 5.1 defines the syntax of $\lambda_{\mathcal{J}}$. A $\lambda_{\mathcal{J}}$ *expression* e is either a constant, a variable, a primitive operation, a lambda abstraction, an application, a creation of an empty object, a property read, or a property assignment. *Variables* x, y, z are drawn from denumerable sets of symbols and *constants* c include JavaScript's primitive values like numbers, strings, booleans, as well as `undefined` and `null`.

To define evaluation, Figure 5.2 defines the semantic domains of $\lambda_{\mathcal{J}}$. Their main component is a *store* σ that maps a *location* l to an *object* o, which is either a native object consisting of a *dictionary* d and a prototype value v, or a function object consisting of a lambda expression $\lambda x.e$ and an environment ρ that binds the free variables. A *dictionary* d models the properties of an object. It maps a constant c to a value v. An *environment* ρ maps a variable x to a value v. A *value* v is either a constant c or a location l.

The syntax of $\lambda_{\mathcal{J}}$ does not make proxies available to the user but offers an internal method to wrap objects. Objects and proxy-objects are always modeled by their meta-data.

5.2 Contracts and Contracted $\lambda_{\mathcal{J}}$

Figure 5.3 defines the syntax of $\lambda_{\mathcal{CON}}$ as an extension of $\lambda_{\mathcal{J}}$. It first introduces constructs for contracts in general, and second, it adds new terms specific to contract monitoring. For simplicity, we focus only on constructors for TreatJS core contracts, but other contracts can be added in the same way or stated in terms of a core contract.

The only new source expressions in $\lambda_{\mathcal{CON}}$ are a *contract assertion* $e@^{\ell} f$ and a *contract definition* $\mathcal{E}$. The contract assertion $e@^{\ell} f$ evaluates an expression f to a contract $\mathcal{C}$ and attaches $\mathcal{C}$ to the value of expression e. The contract assertion is adorned with a *blame label* ℓ which identifies the top-level assertion of that contract. Each *blame label* is implicitly bound to the context and the subject of that assertion.

A *contract definition* $\mathcal{E}$ is the top-level construct for contracts. It is either the definition of a *flat contract* which consists of a predicate expression $\lambda x.e$, a *contract abstraction* (aka constructor definition) that abstracts a variable x from a contract definition $\mathcal{E}$, a *constructor application* which applies a contract abstraction to an argument value, a *contract mapping* that maps a property name indicated by a constant c to a contract definition $\mathcal{E}$, the definition of a function contract, the definition of a dependent contract, or the intersection or union of two contract definitions.

Constant	$\ni$	c			
Variable	$\ni$	x, y, z			
Expression	$\ni$	e, f, g	$::=$	c	*(constant)*
			$\mid$	x	*(variable)*
			$\mid$	$\texttt{op}\,(e, e)$	*(primitive operation)*
			$\mid$	$\lambda x.e$	*(abstraction)*
			$\mid$	$e\,e$	*(application)*
			$\mid$	$\texttt{new}\ e$	*(object creation)*
			$\mid$	$e\,[e]$	*(property read)*
			$\mid$	$e\,[e] = e$	*(property assignment)*

Figure 5.1 Syntax of $\lambda_{\mathcal{J}}$.

Location	$\ni$	l			
Value	$\ni$	u, v, w	$::=$	c	*(constant)*
			$\mid$	l	*(location)*
Dictionary	$\ni$	d	$::=$	$\varnothing$	*(empty set)*
			$\mid$	$d[c \mapsto v]$	*(dictionary extension)*
Object	$\ni$	o	$::=$	$\langle d, v \rangle$	*(native object)*
			$\mid$	$\langle \rho, \lambda x.e \rangle$	*(function object)*
Environment	$\ni$	ρ	$::=$	$\varnothing$	*(empty set)*
			$\mid$	$\rho[x \mapsto v]$	*(environment extension)*
Store	$\ni$	σ	$::=$	$\varnothing$	*(empty set)*
			$\mid$	$\sigma[l \mapsto o]$	*(store extension)*

Figure 5.2 Semantic domains of $\lambda_{\mathcal{J}}$.

Expression	$\ni$	e, f, g	$+=$	$e@^{\ell} f$	*(top-level assertion)*
			$\mid$	$\mathcal{E}$	*(contract definition)*
Contract Definition	$\ni$	$\mathcal{E}, \mathcal{F}, \mathcal{G}$	$::=$	$\texttt{flat}\,(\lambda x.e)$	*(flat contract)*
			$\mid$	$\Lambda x.\mathcal{E}$	*(abstraction)*
			$\mid$	$\mathcal{E}\, e$	*(application)*
			$\mid$	$\mathcal{M}$	*(object contract)*
			$\mid$	$\mathcal{E} \to \mathcal{E}$	*(function contract)*
			$\mid$	$\bullet \mapsto \mathcal{E}$	*(dependent contract)*
			$\mid$	$\mathcal{E} \cap \mathcal{E}$	*(intersection)*
			$\mid$	$\mathcal{E} \cup \mathcal{E}$	*(union)*
Contract Mapping	$\ni$	$\mathcal{M}$	$::=$	$\cdot$	*(empty set)*
			$\mid$	$\mathcal{M}; c : \mathcal{E}$	*(contract binding)*

Figure 5.3 Syntax extension of $\lambda_{\mathcal{CON}}$.

Contract	$\ni$	$\mathcal{C}, \mathcal{D}$	$::=$	$\mathcal{I}$	*(immediate contract)*
			$\mid$	$\mathcal{Q}$	*(delayed contract)*
			$\mid$	$\mathcal{I} \cap \mathcal{C}$	*(intersection)*
			$\mid$	$\mathcal{C} \cup \mathcal{Q}$	*(union)*
Immediate Contract	$\ni$	$\mathcal{I}, \mathcal{J}$	$::=$	$\texttt{flat}\,(l)$	*(flat contract)*
Delayed Contract	$\ni$	$\mathcal{Q}, \mathcal{R}$	$::=$	$\mathcal{O}$	*(object contract)*
			$\mid$	$\mathcal{C} \to \mathcal{D}$	*(function contract)*
			$\mid$	$\bullet \mapsto \mathcal{A}$	*(dependent contract)*
			$\mid$	$\mathcal{Q} \cap \mathcal{R}$	*(intersection)*
Object Contract	$\ni$	$\mathcal{O}$	$::=$	$\cdot$	*(empty set)*
			$\mid$	$\mathcal{O}; c : \mathcal{C}$	*(contract binding)*
Contract Constructor	$\ni$	$\mathcal{A}$	$::=$	$(\rho, \Lambda x.\mathcal{E})$	*(constructor)*

Figure 5.4 Core contracts of $\lambda_{\mathcal{CON}}$.

Contract definitions $\mathcal{E}$ evaluate to a core contract (*contract value*). A *core contract* $\mathcal{C}$ (Figure 5.4) is either an *immediate contract* $\mathcal{I}$, a *delayed contract* $\mathcal{Q}$, an *intersection* of an immediate contract and a contract $\mathcal{I} \cap \mathcal{C}$, or a *union* of two contracts $\mathcal{C} \cup \mathcal{D}$. Core contracts distinguish immediate contracts $\mathcal{I}$ that are checked right away when asserted to a value and delayed contracts $\mathcal{Q}$ that stay on a value until the value is used.

Immediate contracts $\mathcal{I}$ and $\mathcal{J}$ stand for flat contracts defined by a function object l that is interpreted as a predicate on its argument value x. The assertion of a flat contract to a subject value applies the predicate function to the subject value and interprets the outcome of the predicate as a truth value. The assertion of a flat contract always returns the subject value itself.

A delayed contract $\mathcal{Q}$ is either an *object contract* $\mathcal{O}$, a *function contract* $\mathcal{C} \to \mathcal{D}$, a *dependent contract* $\bullet \mapsto \mathcal{A}$, or a finite *intersection* of delayed contracts $\mathcal{Q} \cap \mathcal{R}$.

An *object contract* $\mathcal{O}$ is a sequence of property names c and their contracts $\mathcal{C}$. A ; (semicolon) operator extends an object contract with a new key/contract pair on the right. For technical reasons, we assume that the property name c of the new binding is different from the property names in $\mathcal{O}$. The assertion of an object contract expects that the contract for a property name c holds for property c of the contracted object.

A *function contract* $\mathcal{C} \to \mathcal{D}$ is built from a domain contract $\mathcal{C}$ and range contract $\mathcal{D}$. Calling a function with a function contract asserts $\mathcal{C}$ to the argument (domain) and $\mathcal{D}$ to the result (range) of that function. Both contracts can be an arbitrary other contract.

A *dependent contract* $\bullet \mapsto \mathcal{A}$ is a special kind of a function contract. It is defined by a contract constructor $\mathcal{A}$ that maps the argument value of a function to a contract definition, which is applied to the function's return value.

A *contract constructor* $(\rho, \Lambda x.\mathcal{E})$ is not a contract by itself, but it abstracts a parameter x that is bound in the contract arising from contract definition $\mathcal{E}$. Dependent contracts and constructor applications rely on contract constructors to substitute values for x at runtime.

Delayed contracts include intersections of delayed contracts because for each use of the contracted value the context can choose to fulfill either of them. Delayed contracts stay with the value until the value is used in either an application or a property access.

Without loss of generality, we require core contracts to be in a *canonical form*. The canonical form restricts top-level intersections to an intersection of an immediate contract and a rest contract. This requires that unions and flat contracts are distributivity pulled out of an intersection of delayed contracts such that all immediate parts can be checked right away when asserted to a value and that intersections and unions evaluate in the right way.

5.3 Constraints

The presence of intersection and union contracts requires that a failing contract must not signal a contract violation immediately. A violation may depend on a combination of failures in different sub-contracts. Thus, contract monitoring must connect each contract with the enclosing contract operation, and it must compute a violation in terms of its constituents.

This connection is modeled by so-called *constraints* φ (Figure 5.5). They are tied to a particular contract assertion and link each contract to its next enclosing operation, or at the top-level to its assertion. A constraint φ is either a flat constraint $\flat \blacktriangleleft b$ that maps the outcome of a flat contract b to blame identifier $\flat$ or a constraint that chains the outcome of one or more contracts to the outcome of another contract in $\flat$. There is one constraint for each operator on contracts, namely an indirection constraint $\flat \blacktriangleleft \iota$, a flat constraint $\flat \blacktriangleleft b$, an inversion constraint $\flat \blacktriangleleft \neg\iota$, a function constraint $\flat \blacktriangleleft \iota \to \iota$, an intersection constraint

Identifier	$\ni$	$\flat$	$::=$	ℓ	*(label)*
			$\mid$	ι	*(variable)*
Constraint	$\ni$	φ	$::=$	$\flat \blacktriangleleft \iota$	*(indirection constraint)*
			$\mid$	$\flat \blacktriangleleft b$	*(flat constraint)*
			$\mid$	$\flat \blacktriangleleft \neg\iota$	*(inversion constraint)*
			$\mid$	$\flat \blacktriangleleft \iota \to \iota$	*(function constraint)*
			$\mid$	$\flat \blacktriangleleft \iota \cap \iota$	*(intersection constraint)*
			$\mid$	$\flat \blacktriangleleft \iota \cup \iota$	*(union constraint)*
Constraint List	$\ni$	ς	$::=$	$\cdot$	*(empty list)*
			$\mid$	$\varphi : \varsigma$	*(list extension)*

Figure 5.5 Syntax of constraints.

$\flat \blacktriangleleft \iota \cap \iota$, and a union constraint $\flat \blacktriangleleft \iota \cup \iota$. The inversion constraint handles the reverse of the responsibilities after writing a property on an object with an object contract.

Constraints contain blame identifiers $\flat$ drawn from an unspecified denumerable set of blame labels ℓ and *blame variables* ι. Blame labels ℓ only occur in source programs and identify the top-level assertion of a contract, whereas blame variables ι arise during contract evaluation and identify the assertion of a particular constituent. Each identifier is related to one specific contract assertion in a program. There is at least one identifier for each contract assertion and there may be multiple identifiers for delayed contracts.

During the evaluation, a *constraint list* ς collects constraints that arise from contract monitoring. Technically, each constraint list is a forest which is extended on its leafs. The sequence of elements in the list reflects the temporal order in which the constraints were generated during evaluation. The latest, youngest, constraint is always on top of the list. Blame calculation always happens from this list of constraints.

5.4 Contract Monitoring

This section presents the formal semantics of $\lambda_{\mathcal{CON}}$. The formalization employs *pretty-big-step* semantics [15] to model side effects and signaling of violations while keeping the number of evaluation rules manageable.

A pretty-big-step semantics introduces intermediate term t (Figure 5.6) to model partially evaluated expressions. An *intermediate terms* t is thus an expression with zero or more top-level sub-expressions replaced by their outcomes, which we call *behaviors*. A behavior b is either a value v, a contract $\mathcal{C}$, a contract constructor $\mathcal{A}$, or an *error* `err`, which is either a positive blame $\texttt{+blame}^{\ell}$ or a negative blame $\texttt{-blame}^{\ell}$. New to the syntax of intermediate terms is an internal contract assertion $v@^{\iota}\mathcal{C}$. Terms are constructed with a specific evaluation order in mind and do not occur in source programs. They only arise during evaluation. Moreover, we extend intermediate terms with terms that contain other intermediate terms. This is required to introduce contracts without a big effort.

A *proxy object* (Figure 5.7) is a single location l controlled by a proxy handler that mediates the access to the target location. Proxy objects pack a delayed contract together with its subject value and implement sandbox wrappers that encapsulate data structures when using values inside of predicate code. For simplification, we represent handler objects

Error	$\ni$	$\mathtt{err}$	$::=$	$\mathtt{+blame}^{\ell}$	*(positive blame)*
			$\mid$	$\mathtt{-blame}^{\ell}$	*(negative blame)*
Behavior	$\ni$	b	$::=$	v	*(value)*
			$\mid$	$\mathcal{C}$	*(contract)*
			$\mid$	$\mathcal{A}$	*(constructor)*
			$\mid$	$\mathtt{err}$	*(error)*
Term	$\ni$	t	$::=$	e	*(expression)*
			$\mid$	$\mathtt{op}\,(b, e) \mid \mathtt{op}\,(v, b)$	*(primitive operation)*
			$\mid$	$b\,e \mid l\,b$	*(application)*
			$\mid$	$\mathtt{new}\ b$	*(object creation)*
			$\mid$	$b\,[e] \mid l\,[b]$	*(property read)*
			$\mid$	$b\,[e] = e \mid l\,[b] = e \mid l\,[c] = b$	*(property assignment)*
			$\mid$	$b@^{\ell} f \mid v@^{\ell} b$	*(top-level assertion)*
			$\mid$	$b@^{\iota}\mathcal{C}$	*(contract assertion)*
			$\mid$	$\mathcal{A}\,b$	*(constructor application)*
			$\mid$	$\mathcal{Q}; c : \mathcal{E} \mid \mathcal{O}; c : b$	*(object contract)*
			$\mid$	$b \to \mathcal{E} \mid \mathcal{C} \to b$	*(function contract)*
			$\mid$	$\bullet \mapsto b$	*(dependent contract)*
			$\mid$	$b \cap \mathcal{E} \mid \mathcal{C} \cap b$	*(intersection contract)*
			$\mid$	$b \cup \mathcal{E} \mid \mathcal{C} \cup b$	*(union contract)*
			$\mid$	$t@^{\iota}\mathcal{C} \mid v@^{\iota} t$	*(contract assertion)*
			$\mid$	$t\,t \mid v\,t \mid t\,v$	*(application)*
			$\mid$	$t\,[c]$	*(property read)*
			$\mid$	$t\,[c] = v \mid l\,[c] = t$	*(property assignment)*

Figure 5.6 Intermediate terms of $\lambda_{\mathcal{CON}}$.

Object	$\ni$	o	$+=$	$\langle v, \iota, \mathcal{Q} \rangle$	*(contract proxy)*
			$\mid$	$\langle l, \widehat{\rho} \rangle$	*(sandbox proxy)*
Context	$\ni$	κ	$::=$	$\bullet$	*(global context)*
			$\mid$	$\mathcal{C}$	*(contract)*
			$\mid$	$\mathcal{A}$	*(constructor)*

Figure 5.7 Semantic domains extension of $\lambda_{\mathcal{CON}}$.

by the handler's meta-data. Consequently, $\lambda_{\mathcal{CON}}$ objects are extended with contract proxies $\langle v, \iota, \mathcal{Q} \rangle$ and sandbox proxies $\langle l, \widehat{\rho} \rangle$. A contract proxy is a value wrapped in a delayed contract that is to be monitored when the value is used. A sandbox proxy is a wrapper for another location and mediates the access to that target location.

To state evaluation, a *context* κ distinguishes the global execution context $\bullet$ from the context in a contract $\mathcal{C}$ or a contract constructor $\mathcal{A}$. This context information is needed to blame the correct context for a contract violation.

5.4.1 Evaluation of $\lambda_{\mathcal{CON}}$

The evaluation judgment is similar to a standard big-step evaluation judgment except that its input ranges over intermediate terms and its output is a behavior. The judgment states that the evaluation of term t with initial store σ, constraint list ς, environment ρ, and context κ results in a final heap σ', constraint list ς', and behavior b:

$$\kappa, \rho \ \vdash \ \sigma, \varsigma, t \ \Downarrow \ \sigma', \varsigma', b$$

Figure 5.8 and 5.9 defines the standard evaluation rules for expressions e in $\lambda_{\mathcal{CON}}$. The evaluation rules for expressions are mostly standard. Each rule for a composite expression evaluates exactly one sub-expression and then recursively invokes the evaluation judgment to continue. Once all sub-expressions are evaluated, the respective rule performs the desired operation. The corresponding straightforward error propagation rules are disjoint to the remaining rules because they fire only if an intermediate-term contains an error. Figure 5.10 defines the evaluation rules for intermediate terms with nested terms, and the rule set in Figure 5.11 defines error handling, which always propagates an error to the top level.

The rules Const, Var, and Undef are standard. Rule Op applies a primitive operation to its argument values. The extended set of objects requires to revisit primitive operations on values. Function $\lfloor v \rfloor_\sigma$ (Figure 5.12) first erases all contract monitors and sandbox wrappers from its argument values before it proceeds with the usual operation. This unwrapping makes proxies transparent with respect to primitive operations on locations.

Rule Abs allocates a new function object consisting of a closure with the current environment and rule New allocates a new native object based on the evaluated prototype.

Function application, property lookup, and property assignment distinguish two cases: either the operation applies directly to a target object (non-proxy object) or it applies to a proxy. If the given reference is a non-proxy object, then the usual rules apply: App for function application, Get for property lookup, and Put for property assignment. The evaluation rules for the non-standard cases are given in Section 5.4.5 and Section 5.4.6.

The rules Get, Get-Proto, and Get-Undef implement JavaScript's lookup operation through prototype chains. Each object has an internal prototype property which links to another object or to a constant value, signaling the end of the chain. If a property is not contained in the object's dictionary rule Get-Proto forwards the lookup to the prototype object. When reaching the end of the chain rule Get-Undef returns `undefined` by default.

It remains to define contract assertion. The contract assertion (Figure 5.13) applies after the first sub-expression e evaluates to a value v and the contract expression f evaluates to a contract $\mathcal{C}$. Rule Assert creates a new blame variable ι for each new instantiation of a contract in the source program. It further extends the constraint set by a new constraint that links the outcome of the contract monitor to blame label ℓ.

$$\textsc{Const}\quad \kappa, \rho \vdash \sigma, \varsigma, c \Downarrow \sigma, \varsigma, c$$

$$\textsc{Var}\quad \frac{x \in dom(\rho)}{\kappa, \rho \vdash \sigma, \varsigma, x \Downarrow \sigma, \varsigma, \rho(x)}$$

$$\textsc{Undef}\quad \frac{x \notin dom(\rho)}{\kappa, \rho \vdash \sigma, \varsigma, x \Downarrow \sigma, \varsigma, \texttt{undefined}}$$

$$\textsc{Op-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \texttt{op}(b, f) \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \texttt{op}(e, f) \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Op-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, f \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \texttt{op}(v, b) \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \texttt{op}(v, f) \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Op}\quad \frac{v' = \lfloor v \rfloor_\sigma \qquad w' = \lfloor w \rfloor_\sigma \qquad v', w' \in dom(\delta_{op}) \qquad u = \delta_{op}(v', w')}{\kappa, \rho \vdash \sigma, \varsigma, \texttt{op}(v, w) \Downarrow \sigma, \varsigma, u}$$

$$\textsc{Abs}\quad \frac{l \notin dom(\sigma) \qquad \sigma' = \sigma[l \mapsto \langle \rho, \lambda x.e \rangle]}{\kappa, \rho \vdash \sigma, \varsigma, \lambda x.e \Downarrow \sigma', \varsigma, l}$$

$$\textsc{App-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma, \varsigma'', b\, f \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, e\, f \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{App-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, f \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', l\, b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\, f \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{App}\quad \frac{\langle \rho', \lambda x.e \rangle = \sigma(l) \qquad \rho'[x \mapsto v] \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', w}{\kappa, \rho \vdash \sigma, \varsigma, l\, v \Downarrow \sigma', \varsigma', w}$$

Figure 5.8 Evaluation rules for intermediate terms of $\lambda_{\mathcal{CON}}$.

$$\textsc{New-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \texttt{new}\ b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \texttt{new}\ e \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{New}\quad \frac{l \notin dom(\sigma) \qquad \sigma' = \sigma[l \mapsto \langle \varnothing, v \rangle]}{\kappa, \rho \vdash \sigma, \varsigma, \texttt{new}\ v \Downarrow \sigma', \varsigma, l}$$

$$\textsc{Get-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\,[f] \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, e\,[f] \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Get-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, f \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', l\,[b] \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\,[f] \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Get}\quad \frac{\langle d, u \rangle = \sigma(l) \qquad c \in dom(d)}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] \Downarrow \sigma, \varsigma, d(c)}$$

$$\textsc{Get-Proto}\quad \frac{\langle d, l' \rangle = \sigma(l) \qquad c \notin dom(d) \qquad \kappa, \rho \vdash \sigma, \varsigma, l'\,[c] \Downarrow \sigma', \varsigma', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] \Downarrow \sigma', \varsigma', b'}$$

$$\textsc{Get-Undef}\quad \frac{\langle d, c' \rangle = \sigma(l) \qquad c \notin dom(d)}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] \Downarrow \sigma, \varsigma, \texttt{undefined}}$$

$$\textsc{Put-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, e \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\,[f] = g \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, e\,[f] = g \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, f \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', l\,[b] = g \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\,[f] = g \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put-G}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, g \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', l\,[c] = b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] = g \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put}\quad \frac{\langle d, w \rangle = \sigma(l) \qquad \sigma' = \sigma[l \mapsto \langle d[c \mapsto v], w \rangle]}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] = v \Downarrow \sigma', \varsigma, v}$$

Figure 5.9 Evaluation rules for intermediate terms of $\lambda_{\mathcal{CON}}$ (cont'd).

$$\textsc{App-T-1}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\, t_2 \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, t_1\, t_2 \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{App-T-2}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\, v \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, t\, v \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{App-T-3}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', v\, b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, v\, t \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Get-T}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\,[c] \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, t\,[c] \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put-T-1}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b\,[c] = v \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, t\,[c] = v \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put-T-2}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', l\,[c] = b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, l\,[c] = t \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Assert-T-1}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b@^{\iota}\mathcal{C} \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, t@^{\iota}\mathcal{C} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Assert-T-2}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, t \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', v@^{\iota}b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, v@^{\iota}t \Downarrow \sigma'', \varsigma'', b'}$$

Figure 5.10 Evaluation rules for intermediate terms of $\lambda_{\mathcal{CON}}$ (cont'd).

$$\textsc{Error-Op-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{op}\,(\mathtt{err},f) \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Op-F}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{op}\,(v,\mathtt{err}) \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-App-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,f \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-App-F}\quad \kappa,\rho \vdash \sigma,\varsigma,l\,\mathtt{err} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-App-T-1}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,t \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-App-T-2}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,v \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-New-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{new}\ \mathtt{err} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Get-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,[f] \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Get-F}\quad \kappa,\rho \vdash \sigma,\varsigma,l\,[\mathtt{err}] \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Get-T}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,[\mathcal{C}] \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Put-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,[f] = g \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Put-F}\quad \kappa,\rho \vdash \sigma,\varsigma,l\,[\mathtt{err}] = g \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Put-G}\quad \kappa,\rho \vdash \sigma,\varsigma,l\,[c] = \mathtt{err} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Put-T}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}\,[c] = v \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Assert-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}@^{\ell}\mathcal{E} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Assert-F}\quad \kappa,\rho \vdash \sigma,\varsigma,v@^{\ell}\mathtt{err} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Assert-T}\quad \kappa,\rho \vdash \sigma,\varsigma,v@^{\iota}\mathtt{err} \Downarrow \sigma,\varsigma,\mathtt{err}$$

$$\textsc{Error-Assert}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathtt{err}@^{\iota}\mathcal{C} \Downarrow \sigma,\varsigma,\mathtt{err}$$

Figure 5.11 Evaluation rules for error handling of $\lambda_{\mathcal{CON}}$.

$$\lfloor v \rfloor_{\sigma} = \begin{cases} \lfloor w \rfloor_{\sigma} & v = l, \langle w, \iota, \mathcal{Q} \rangle = \sigma\,(l) \\ \lfloor l' \rfloor_{\sigma} & v = l, \langle l', \widehat{\rho} \rangle = \sigma\,(l) \\ v & \text{otherwise} \end{cases}$$

Figure 5.12 Unwrap proxy objects.

$$\textsc{Assert-E}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,e \Downarrow \sigma',\varsigma',b \qquad \kappa,\rho \vdash \sigma',\varsigma',b@^{\ell}f \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,e@^{\ell}f \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Assert-F}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,f \Downarrow \sigma,\varsigma,b \qquad \kappa,\rho \vdash \sigma,\varsigma,v@^{\ell}b \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,v@^{\ell}f \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Assert}\quad \frac{\iota \notin dom\,(\varsigma) \qquad \kappa,\rho \vdash \sigma,(\ell \blacktriangleleft \iota) : \varsigma, v@^{\iota}\mathcal{C} \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,v@^{\ell}\mathcal{C} \Downarrow \sigma'',\varsigma'',b'}$$

Figure 5.13 Evaluation rules for top-level contract assertion.

$$\textsc{Constructor-Abstraction}\quad \bullet, \rho \vdash \sigma, \varsigma, \Lambda x.\mathcal{E} \Downarrow \sigma', \varsigma', (\varnothing, \Lambda x.\mathcal{E})$$

$$\textsc{Constructor-Abstraction-Sandbox}\quad \mathcal{A}, \widehat{\rho} \vdash \sigma, \varsigma, \Lambda x.\mathcal{E} \Downarrow \sigma, \varsigma, (\widehat{\rho}, \Lambda x.\mathcal{E})$$

$$\textsc{Constructor-Application-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E} \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma, \varsigma', b\, f \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E}\, f \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Constructor-Application-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, f \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \mathcal{A}\, b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{A}\, f \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Constructor-Application}\quad \frac{(\widehat{\rho}, \Lambda x.\mathcal{E}) = \mathcal{A} \qquad \mathit{wrap}\,(v, \widehat{\rho}, \sigma, \varsigma) = (\widehat{v}, \sigma', \varsigma') \qquad \mathcal{A}, \widehat{\rho}[x \mapsto \widehat{v}] \vdash \sigma', \varsigma', \mathcal{E} \Downarrow \sigma'', \varsigma'', b}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{A}\, v \Downarrow \sigma'', \varsigma'', b}$$

Figure 5.14 Evaluation rules for contract definition.

5.4.2 Contract Definition

Contracts $\mathcal{C}$ are built from contract definitions $\mathcal{E}$ that evaluate to a canonical contract value. Figure 5.14, 5.15, and 5.16 define the evaluation of contract definitions $\mathcal{E}$. Again, each rule evaluates one sub-expression and applies the evaluation judgment recursively until all sub-expressions are evaluated to a canonical contract. Figure 5.17 shows the corresponding error propagation rules.

Contract definition happens in the context of a secure sandbox environment to prevent inference with the actual program execution. For clarity, we write $\widehat{\rho}$ for a (secure) sandbox environment, $\widehat{u}$, $\widehat{v}$ and $\widehat{w}$ for a secure value, and $\widehat{l}$ for a secure location. A secure value $\widehat{v}$ is either a constant or a secure location $\widehat{l}$, which is a location of a sandbox proxy or an object created inside of a sandbox environment. A sandbox environment $\widehat{\rho}$ is an environment ρ that maps a variable x to a secure $\widehat{v}$.

A contract abstraction (constructor definition) $\Lambda x.\mathcal{E}$ (Rule Constructor-Abstraction and Constructor-Abstraction-Sandbox) evaluates to a contract closure $\mathcal{A}$ containing the abstraction together with an empty (secure) environment or together with a sandbox environment when defining the abstraction inside of a sandbox environment.

The constructor application (Rule Constructor-Application) starts after the first expression evaluates to a contract closure, and the second expression evaluates to a value. It wraps the given value v and evaluates contract definition $\mathcal{E}$ in the sandbox environment binding the sandboxed value $\widehat{v}$. Sandbox environments are always built like this. Meta-function $\mathit{wrap}\,(v, \widehat{\rho}, \sigma, \varsigma)$ (cf. Section 5.4.6) packs the value in a sandbox proxy to avoid interference.

Rule Define-Flat and Define-Flat-Sandbox defines a flat contract. The contained predicate definition evaluates to a function closure containing the predicate. The function closure contains either an empty environment or a sandbox environment.

The remaining construction rules for composed contracts are straightforward. An object contract is built from the key/contract bindings of a contract mapping, a function contract is built from a contract for the range and a contract for the domain of a function, a dependent contract is built from a contract constructor, and the intersection and union of two contracts is built from two sub-contract.

$$\textsc{Define-Flat}\quad \frac{\bullet, \emptyset \vdash \sigma, \varsigma, \lambda x.e \Downarrow \sigma', \varsigma, l}{\bullet, \rho \vdash \sigma, \varsigma, \texttt{flat}\,(\lambda x.e) \Downarrow \sigma', \varsigma, \texttt{flat}\,(l)}$$

$$\textsc{Define-Flat-Sandbox}\quad \frac{\mathcal{A}, \widehat{\rho} \vdash \sigma, \varsigma, \lambda x.e \Downarrow \sigma', \varsigma, l}{\mathcal{A}, \widehat{\rho} \vdash \sigma, \varsigma, \texttt{flat}\,(\lambda x.e) \Downarrow \sigma', \varsigma, \texttt{flat}\,(l)}$$

$$\textsc{Define-Object-M}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{M} \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b; c : \mathcal{E} \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{M}; c : \mathcal{E} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Define-Object-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E} \Downarrow \sigma', \varsigma', b \qquad \widehat{\rho} \vdash \sigma', \varsigma', \mathcal{O}; c : b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{O}; c : \mathcal{E} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Define-Object}\quad \kappa, \rho \vdash \sigma, \varsigma, \mathcal{O} \Downarrow \sigma, \varsigma, \mathcal{O}$$

$$\textsc{Define-Function-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E} \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', b \to \mathcal{F} \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E} \to \mathcal{F} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Define-Function-F}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{F} \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \mathcal{C} \to b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{C} \to \mathcal{F} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Define-Function}\quad \kappa, \rho \vdash \sigma, \varsigma, \mathcal{C} \to \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{C} \to \mathcal{D}$$

$$\textsc{Define-Dependent-E}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{E} \Downarrow \sigma', \varsigma', b \qquad \kappa, \rho \vdash \sigma', \varsigma', \bullet \mapsto b \Downarrow \sigma'', \varsigma'', b'}{\kappa, \rho \vdash \sigma, \varsigma, \bullet \mapsto \mathcal{E} \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Define-Dependent}\quad \kappa, \rho \vdash \sigma, \varsigma, \bullet \mapsto \mathcal{A} \Downarrow \sigma, \varsigma, \bullet \mapsto \mathcal{A}$$

Figure 5.15 Evaluation rules for contract definition (cont'd).

5.4.3 Contract Normalization

Contracts $\mathcal{C}$ are created by a set of contract expression $\mathcal{E}$ that may also contain arbitrary top-level intersections of contracts. However, contract monitoring requires contracts to be in a *canonical form* which separates immediate and delayed parts and which enables to evaluate contracts in the correct order. To this end, contract evaluation *normalizes* non-canonical contracts into a canonical form before it starts their enforcement.

The rules in Figure 5.18 implement contract normalization. The rules are disjoint to the already existing contract definitions in Figure 5.16 and fire only on non-canonical contracts.

Rule N-Immediate, N-Intersection, and N-Union interchange the operands of an intersection. Rule N-Factorize distributively pulls a union contract out of an intersection and rule N-Rearrange rearranges the parenthesis of a nested intersection. As we always build contracts from canonical contracts, we only need to consider the non-canonical intersection of two arbitrary other (canonical) contracts.

Each normalization rule, except rule N-Factorize, preserves the total number contract checks. They only reorganize the contained contracts. Only rule N-Factorize builds a new union out of two intersection contracts and thus it duplicates the contract $\mathcal{D}$. This is required

$$\textsc{Define-Intersection-E}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{E} \Downarrow \sigma',\varsigma',b \qquad \kappa,\rho \vdash \sigma',\varsigma',b \cap \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{E} \cap \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Define-Intersection-F}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{F} \Downarrow \sigma',\varsigma',b \qquad \kappa,\rho \vdash \sigma',\varsigma',\mathcal{C} \cap b \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \cap \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Define-Intersection}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{I} \cap \mathcal{C} \Downarrow \sigma,\varsigma,\mathcal{I} \cap \mathcal{C}$$

$$\textsc{Define-Delayed-Intersection}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{Q} \cap \mathcal{R} \Downarrow \sigma,\varsigma,\mathcal{Q} \cap \mathcal{R}$$

$$\textsc{Define-Union-E}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{E} \Downarrow \sigma',\varsigma',b \qquad \kappa,\rho \vdash \sigma',\varsigma',b \cup \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{E} \cup \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Define-Union-F}\quad \frac{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{F} \Downarrow \sigma',\varsigma',b \qquad \kappa,\rho \vdash \sigma',\varsigma',\mathcal{C} \cup b \Downarrow \sigma'',\varsigma'',b'}{\kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \cup \mathcal{F} \Downarrow \sigma'',\varsigma'',b'}$$

$$\textsc{Define-Union}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \cup \mathcal{D} \Downarrow \sigma,\varsigma,\mathcal{C} \cup \mathcal{D}$$

Figure 5.16 Evaluation rules for contract definition (cont'd).

$$\textsc{Error-Constructor-Application-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{A}\,\texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Object-M}\quad \kappa,\rho \vdash \sigma,\varsigma,\texttt{err};c:\mathcal{E} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Object-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{O};c:\texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Function-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\texttt{err} \to \mathcal{F} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Function-F}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \to \texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Dependent-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\bullet \mapsto \texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Intersection-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\texttt{err} \cap \mathcal{F} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Intersection-F}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \cap \texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Union-E}\quad \kappa,\rho \vdash \sigma,\varsigma,\texttt{err} \cup \mathcal{F} \Downarrow \sigma,\varsigma,\texttt{err}$$

$$\textsc{Error-Define-Union-F}\quad \kappa,\rho \vdash \sigma,\varsigma,\mathcal{C} \cup \texttt{err} \Downarrow \sigma,\varsigma,\texttt{err}$$

Figure 5.17 Evaluation rules for error handling of $\lambda_{\mathcal{COM}}$ (cont'd).

$$\textsc{N-Immediate}\quad \kappa, \rho \vdash \sigma, \varsigma, \mathcal{Q} \cap \mathcal{I} \Downarrow \sigma, \varsigma, \mathcal{I} \cap \mathcal{Q}$$

$$\textsc{N-Intersection}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, (\mathcal{I} \cap \mathcal{C}) \cap \mathcal{Q} \Downarrow \sigma, \varsigma, \mathcal{D}}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{Q} \cap (\mathcal{I} \cap \mathcal{C}) \Downarrow \sigma, \varsigma, \mathcal{D}}$$

$$\textsc{N-Union}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, (\mathcal{C} \cup \mathcal{D}) \cap \mathcal{Q} \Downarrow \sigma, \varsigma, \mathcal{C}'}{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{Q} \cap (\mathcal{C} \cup \mathcal{D}) \Downarrow \sigma, \varsigma, \mathcal{C}'}$$

$$\textsc{N-Factorize}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{C}_1 \cap \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{C}' \qquad \kappa, \rho \vdash \sigma, \varsigma, \mathcal{C}_2 \cap \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{D}'}{\kappa, \rho \vdash \sigma, \varsigma, (\mathcal{C}_1 \cup \mathcal{C}_2) \cap \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{C}' \cup \mathcal{D}'}$$

$$\textsc{N-Rearrange}\quad \frac{\kappa, \rho \vdash \sigma, \varsigma, \mathcal{C} \cap \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{C}'}{\kappa, \rho \vdash \sigma, \varsigma, (\mathcal{I} \cap \mathcal{C}) \cap \mathcal{D} \Downarrow \sigma, \varsigma, \mathcal{I} \cap \mathcal{C}'}$$

Figure 5.18 Evaluation rules for contract normalization.

to preserve the semantics of the contract, but it increases the total number of contract checks.

The logical justification for all these rules is the associativity, commutativity, and distributivity of intersection and union contract. To guarantee this, a compatibility check (cf. Section 5.6) ensures that the contract assertion does not mix up contracts that belong to different operands of an intersection or union contract.

5.4.4 Contract Assertion

Contract assertion starts after evaluating the first expression to a subject value v and evaluating the contract definition to a contract $\mathcal{C}$. Figure 5.19 shows its evaluation rules.

Rule Flat starts evaluating a flat contract by applying the predicate to the wrapped subject value. Meta-function $drop\,(v, \iota, \sigma, \varsigma)$ first drops all non-compatible delayed contracts (cf. Section 5.6) on the subject value. It removes all contract wrappers that do not belong to the same side of an intersection or union. Function *wrap* wraps the resulting value in a sandbox proxy to avoid interference (cf. Section 5.4.6). Afterwards, it triggers the predicate evaluation on the wrapped subject value. All operations happen under a new context, which is the current flat contract. This ensures that all contract violations that happen during the predicate evaluation blame the contract, instead of the global context.

The rules Intersection and Union implement contract checking of top-level intersection and union contracts. Both rules assert the first sub-contract to the subject value and assert the second sub-contract to the resulting behavior of the first assertion. The contract evaluation installs a new constraint that links the outcome of both sub-contracts to the satisfaction of the contract.

Finally, rule Delayed wraps a subject value and a delayed contract together with the associated blame identifiers in a contract proxy that later checks the contract when the value is used (see also Section 5.4.5).

FLAT

$$\frac{\langle \widehat{\rho}, \lambda x.e \rangle = \sigma\,(l) \qquad \mathit{drop}\,(v, \iota, \sigma, \varsigma) = (w, \sigma') \qquad \mathit{wrap}\,(w, \widehat{\rho}, \sigma', \varsigma) = (\widehat{w}, \sigma'', \varsigma') \qquad \texttt{flat}\,(l), \rho \;\vdash\; \sigma'', \varsigma', v@^{\iota}\,(l\,\widehat{w}) \;\Downarrow\; \sigma''', \varsigma'', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, v@^{\iota}\texttt{flat}\,(l) \;\Downarrow\; \sigma''', \varsigma'', b}$$

UNION

$$\frac{\iota_1, \iota_2 \notin \mathit{dom}\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \cup \iota_2) : \varsigma, (v@^{\iota_1}\mathcal{C})\,@^{\iota_2}\mathcal{D} \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, v@^{\iota}(\mathcal{C} \cup \mathcal{D}) \;\Downarrow\; \sigma', \varsigma', b}$$

INTERSECTION

$$\frac{\iota_1, \iota_2 \notin \mathit{dom}\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, (v@^{\iota_1}\mathcal{I})\,@^{\iota_2}\mathcal{C} \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, v@^{\iota}(\mathcal{I} \cap \mathcal{C}) \;\Downarrow\; \sigma', \varsigma', b}$$

DELAYED

$$\frac{l' \notin \mathit{dom}\,(\sigma) \qquad \sigma' = \sigma\,[l' \mapsto \langle l, \iota, \mathcal{Q} \rangle]}{\kappa, \rho \;\vdash\; \sigma, \varsigma, v@^{\iota}\mathcal{Q} \;\Downarrow\; \sigma', \varsigma, l'}$$

Figure 5.19 Evaluation rules for contract assertion in $\lambda_{\mathcal{CON}}$.

APP-FUNCTION

$$\frac{\langle w, \iota, (\mathcal{C} \to \mathcal{D}) \rangle = \sigma\,(l) \qquad \iota_1, \iota_2 \notin \mathit{dom}\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \to \iota_2) : \varsigma, (w\,(v@^{\iota_1}\mathcal{C}))\,@^{\iota_2}\mathcal{D} \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,v \;\Downarrow\; \sigma', \varsigma', b}$$

APP-DEPENDENT

$$\frac{\langle w, \iota, (\bullet \mapsto \mathcal{A}) \rangle = \sigma\,(l) \qquad \mathit{drop}\,(v, \iota, \sigma, \varsigma) = (w', \sigma') \qquad \kappa, \rho \;\vdash\; \sigma', \varsigma', (w\,v)\,@^{\iota}\,(\mathcal{A}\,w') \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,v \;\Downarrow\; \sigma', \varsigma', b}$$

APP-OBJECT

$$\frac{\langle w, \iota, \mathcal{O} \rangle = \sigma\,(l) \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,v \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,v \;\Downarrow\; \sigma', \varsigma', b}$$

APP-INTERSECTION

$$\frac{\langle w, \iota, (\mathcal{Q} \cap \mathcal{R}) \rangle = \sigma\,(l) \qquad \iota_1, \iota_2 \notin \mathit{dom}\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, ((w@^{\iota_1}\mathcal{Q})\,@^{\iota_2}\mathcal{R})\,v \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,v \;\Downarrow\; \sigma', \varsigma', b}$$

Figure 5.20 Evaluation rules for function application on a contract proxy.

$$\textsc{Get-Function}\quad \frac{\langle w, \iota, (\mathcal{C} \to \mathcal{D})\rangle = \sigma\,(l) \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Get-Dependent}\quad \frac{\langle w, \iota, (\bullet \mapsto \mathcal{A})\rangle = \sigma\,(l) \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Get-Object-Existing}\quad \frac{\langle w, \iota, \mathcal{O}\rangle = \sigma\,(l) \qquad c : \mathcal{C} \in \mathcal{O} \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, (w\,[c])\,@^{\iota}\mathcal{C} \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Get-Object-NotExisting}\quad \frac{\langle w, \iota, \mathcal{O}\rangle = \sigma\,(l) \qquad \nexists c : \mathcal{C} \in \mathcal{O} \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Get-Intersection}\quad \frac{\begin{array}{c}\langle w, \iota, (\mathcal{Q} \cap \mathcal{R})\rangle = \sigma\,(l) \\ \iota_1, \iota_2 \notin dom\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, ((w@^{\iota_1}\mathcal{Q})\,@^{\iota_2}\mathcal{R})\,[c] \;\Downarrow\; \sigma', \varsigma', b\end{array}}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] \;\Downarrow\; \sigma', \varsigma', b}$$

Figure 5.21 Evaluation rules for property read on a contract proxy.

$$\textsc{Put-Function}\quad \frac{\langle w, \iota, (\mathcal{C} \to \mathcal{D})\rangle = \sigma\,(l) \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Put-Dependent}\quad \frac{\langle w, \iota, (\bullet \mapsto \mathcal{A})\rangle = \sigma\,(l) \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Put-Object-Existing}\quad \frac{\begin{array}{c}\langle w, \iota, \mathcal{O}\rangle = \sigma\,(l) \\ c : \mathcal{C} \in \mathcal{O} \qquad \iota_1 \notin dom\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft \neg\iota_1 : \varsigma, w\,[c] = (v@^{\iota_1}\mathcal{C}) \;\Downarrow\; \sigma', \varsigma', b\end{array}}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Put-Object-NotExisting}\quad \frac{\langle w, \iota, \mathcal{O}\rangle = \sigma\,(l) \qquad \nexists c : \mathcal{C} \in \mathcal{O} \qquad \kappa, \rho \;\vdash\; \sigma, \varsigma, w\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}$$

$$\textsc{Put-Intersection}\quad \frac{\begin{array}{c}\langle w, \iota, (\mathcal{Q} \cap \mathcal{R})\rangle = \sigma\,(l) \\ \iota_1, \iota_2 \notin dom\,(\varsigma) \qquad \kappa, \rho \;\vdash\; \sigma, \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, ((w@^{\iota_1}\mathcal{Q})\,@^{\iota_2}\mathcal{R})\,[c] = v \;\Downarrow\; \sigma', \varsigma', b\end{array}}{\kappa, \rho \;\vdash\; \sigma, \varsigma, l\,[c] = v \;\Downarrow\; \sigma', \varsigma', b}$$

Figure 5.22 Evaluation rules for property assignment on a contract proxy.

5.4.5 Delayed Contract Checking

Object contracts, function contracts, dependent contracts, and intersections thereof are delayed contracts. They are dormant and stay with the subject value until the value is used in an application or a property access. Therefore, property read, property write, and function application distinguish two cases: either the operation applies directly to a non-proxy object or it applies to a proxy object. In the former case, the rules in Figure 5.8 and 5.9 apply.

In the non-standard cases, the operation applies to a proxy object which is either a contract proxy or a sandbox proxy. Figure 5.20, 5.21, and 5.22 contain the evaluation rules for function application, property read, and property write on a contract proxy.

Rule App-Function handles the call to a value with a function contract. The domain contract $\mathcal{C}$ is attached to the argument value. Next, the function application proceeds by passing the contracted argument value to the proxy's target location. After completion, the range contract $\mathcal{D}$ is applied to the function's return. In sharp contrast to previous work, the blame propagation is handled indirectly by creating new blame variables for the domain and range part. A new constraint is added that chains the outcome of both sub-contracts to the assertion of the function contract.

Rule App-Dependent handles the call to a value with a dependent contract. In this case, function application proceeds on the proxy's target value. Next, meta-function $drop\,(v, \iota, \sigma, \varsigma)$ removes all contract wrappers that belong to another side of an intersection or union. The argument value gets passed to the contract constructor that returns a contract for the application's return. Finally, the contract assertion proceeds on the application's return.

Rule App-Object handles the call to a value with an object contract. As an object contract does not apply to a function application, it gets ignored and the operation proceeds on the proxy's target value.

Rule App-Intersection handles the call to a value with an intersection contract. It sequentially attaches both sub-contracts to the subject value and proceeds with the application on the contracted value. The generated constraint combines the outcomes of both sub-contracts. Unlike the union contract, a new intersection constraint arises at each use of the contracted value which implements the choice of the context.

A property read and a property write operation on a value with a function or a dependent contract (Rules Get-Function, Get-Dependent, Put-Function, and Put-Dependent) simply ignores the contract and proceeds with the usual operation on the proxy's target value. Both kinds of contracts do not apply to property access.

A property read on a value with an object contract has two cases depending on the presence of a contract for the accessed property. If a contract exists (Rule Get-Object-Existing), then the contract is attached to the value that returns from the access on the proxy's target value. Otherwise (Rule Get-Object-NotExisting), no contract applies and the value is simply returned.

A property assignment on a value with an object contract continues with the usual operation on the proxy's target value after attaching the property's contract to the new value. In TreatJS the adherence of a contract is also checked on assignments, but the check happens in the context of a new constraint that flips the responsibilities. The context of an assignment is responsible to assign a value according to its specification. If no contract exists (Rule Put-Object-NotExisting), then the operation continues with the usual value.

Finally, the rules Get-Intersection and Put-Intersection handle property access on a value with an intersection contract. Like function application, property access sequentially attaches both sub-contracts to the subject value and creates a new constraint to link the outcome of both sub-contract. Then, it continues with the usual operation.

$$\textsc{Wrap-Constant}\quad wrap\,(c,\rho,\sigma,\varsigma) = (c,\sigma,\varsigma)$$

$$\textsc{Wrap-Error}\quad wrap\,(\texttt{err},\rho,\sigma,\varsigma) = (\texttt{err},\sigma,\varsigma)$$

$$\textsc{Wrap-Contract}\quad wrap\,(\mathcal{C},\rho,\sigma,\varsigma) = (\mathcal{C},\sigma,\varsigma)$$

$$\textsc{Wrap-Constructor}\quad wrap\,(\mathcal{A},\rho,\sigma,\varsigma) = (\mathcal{A},\sigma,\varsigma)$$

$$\textsc{Wrap-Object}\quad \frac{\langle d,v\rangle = \sigma\,(l) \qquad \widehat{l} \notin dom\,(\sigma) \qquad \sigma' = \sigma\left[\widehat{l} \mapsto \langle l,\widehat{\rho}\rangle\right]}{wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l},\sigma',\varsigma\right)}$$

$$\textsc{Wrap-Function}\quad \frac{\langle \rho,\lambda x.e\rangle = \sigma\,(l) \qquad \widehat{l} \notin dom\,(\sigma) \qquad \sigma' = \sigma[\widehat{l} \mapsto \langle \widehat{\rho},\lambda x.e\rangle]}{wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l},\sigma',\varsigma\right)}$$

$$\textsc{Wrap-Delayed}\quad \frac{\langle l',\iota,\mathcal{Q}\rangle = \sigma\,(l) \qquad wrap\,(l',\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l},\sigma',\varsigma'\right) \qquad mirror\left(l,\widehat{l},\sigma,\varsigma\right) = \left(\widehat{l'},\sigma'',\varsigma''\right)}{wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l'},\sigma'',\varsigma'\right)}$$

$$\textsc{Wrap-Proxy1}\quad \frac{\langle l,\widehat{\rho}\rangle = \sigma\left(\widehat{l}\right)}{wrap\left(\widehat{l},\widehat{\rho},\sigma,\varsigma\right) = \left(\widehat{l},\sigma,\varsigma\right)}$$

$$\textsc{Wrap-Proxy2}\quad \frac{\langle l,\widehat{\rho}'\rangle = \sigma\left(\widehat{l}\right) \qquad wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l'},\sigma',\varsigma'\right)}{wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l'},\sigma',\varsigma'\right)}$$

$$\textsc{Wrap-Existing}\quad \frac{\exists\widehat{l} \in dom\,(\sigma)\,.\,\langle l,\widehat{\rho}\rangle = \sigma\left(\widehat{l}\right)}{wrap\,(l,\widehat{\rho},\sigma,\varsigma) = \left(\widehat{l},\sigma,\varsigma\right)}$$

Figure 5.23 Sandbox encapsulation.

In contrast to other contract implementations, function contracts, dependent contracts, and object contracts might also be applied to non-function and non-object values. In this case, a primitive value gets wrapped in the contract proxy.

For example, having a function contract on a primitive value leads to two cases: either the value is only used in a primitive operation or the value is used in a function application. In the former case, the function contract gets removed before the primitive operation applies. A contract violation is not possible. In the latter case, the evaluation will obviously get stuck when using a non-function value in an application. However, as the domain contract gets applied to the argument value a contract violation blaming the context may arise before getting stuck.

5.4.6 Sandbox Encapsulation

To guarantee noninterference with the execution of a contract abiding host program, TreatJS wraps all subject values and constructor arguments that are passed to a contract in a proxy membrane. The proxy membrane protects the value and prohibits side-effecting operations.

To wrap a value, the evaluation rules use the built-in meta-function $wrap\,(v,\rho,\sigma,\varsigma)$, which returns a *secure value* $\widehat{v}$ for a given value v. A secure value is either a constant or a location to either an object wrapped in a sandbox proxy or an object defined inside of a secure environment. Figure 5.23 shows its definition.

$$\textit{Error} \ni \texttt{err} \mathrel{+}= \texttt{exn} \quad \textit{(sandbox violation)}$$

Figure 5.24 Intermediate terms extension of $\lambda_{\mathcal{CON}}$.

$$\textsc{Get-Sandbox}\ \frac{\langle l', \widehat{\rho}\rangle = \sigma(l) \qquad \kappa, \rho \vdash \sigma, \varsigma, l'[c] \Downarrow \sigma', \varsigma', b \qquad \mathit{wrap}(b, \widehat{\rho}, \varsigma, \sigma) = (b', \sigma'', \varsigma'')}{\kappa, \rho \vdash \sigma, \varsigma, l[c] \Downarrow \sigma'', \varsigma'', b'}$$

$$\textsc{Put-Sandbox}\ \frac{\langle l', \widehat{\rho}\rangle = \sigma(l)}{\kappa, \rho \vdash \sigma, \varsigma, l[c] = v \Downarrow \sigma, \varsigma, \texttt{exn}}$$

Figure 5.25 Evaluation rules for sandbox operations.

A primitive value (Rule Wrap-Constant), an error (Rule Wrap-Error), a contract (Rule Wrap-Contract), and a contract (Rule Wrap-Constructor) are not wrapped and secure by definition. To wrap a location that points to a native (non-function) object (Rule Wrap-Object), the location is packed in a fresh sandbox proxy along with the current sandbox environment. This packaging ensures that each further access to the wrapped location has to use the current sandbox environment. If the location points to a function object (Rule Wrap-Function) then the function closure gets redefined in our sandbox environment, i.e., the function body is bound to the current sandbox environment. As this step removes all existing bindings of a closure, it requires that needed values are imported into the sandbox environment before. Function application on a sandboxed function object is identical to a normal function application and does not need any special treatment.

Particular consideration must be given to the case when the value is a location that points to a contract proxy (Rule Wrap-Delayed). In this case, the wrap operation must reorganize the contract on the target value according to the selected evaluation semantics. It continues with the wrap operation on the proxy's target value and applies meta-function $\mathit{mirror}\left(l, \widehat{l}, \sigma, \varsigma\right)$ to the original and the new secure location. The function mirrors the contracts on l to $\widehat{l}$ according to the chosen evaluation semantics, which are defined in Section 5.5.

The remaining rules handle wrapping of sandbox proxies. If the given location points to a sandbox proxy for that sandbox environment (Rule Wrap-Proxy1) then the same location is returned. If the location points to a proxy object that is wrapped in another sandbox environment (Rule Wrap-Proxy2), then the contained target location gets re-wrapped in the current sandbox environment. Finally, if there already exists a sandbox proxy for the given location (Rule Wrap-Existing), then the location to the existing proxy gets returned.

Finally, we must define operations on sandbox proxies. To this end, we extend the set of possible error messages by a sandbox violation. $\lambda_{\mathcal{CON}}$ errors now include a sandbox violation `exn` that arises when violating a sandbox constraint. Figure 5.24 shows the extension.

Figure 5.25 shows the evaluation rules for operations on sandbox values. A property read on a sandboxed location continues the operation on the proxy target and wraps the resulting behavior in the current sandbox environment. This wrapping guarantees that after wrapping a location, no unprotected (unwrapped) location is reachable from a given location.

$$\textsc{Lax}\quad mirror_{Lax}\left(l, \widehat{l}, \sigma, \varsigma\right) = \left(\widehat{l}, \sigma, \varsigma\right)$$

$$\textsc{Picky}\quad \frac{\langle l', \iota, \mathcal{Q}\rangle = \sigma\left(l\right) \qquad \widehat{l'} \notin dom\left(\sigma\right) \qquad \sigma' = \sigma\left[\widehat{l'} \mapsto \left\langle \widehat{l}, \iota, \mathcal{Q}\right\rangle\right]}{mirror_{Picky}\left(l, \widehat{l}, \sigma, \varsigma\right) = \left(\widehat{l'}, \sigma', \varsigma\right)}$$

$$\textsc{Indy}\quad \frac{\langle l', \iota, \mathcal{Q}\rangle = \sigma\left(l\right) \qquad \ell, \iota_1 \notin dom\left(\varsigma\right) \qquad \varsigma' = \ell \ltimes \iota \blacktriangleleft \iota_1 : \varsigma \qquad \widehat{l'} \notin dom\left(\sigma\right) \qquad \sigma' = \sigma\left[\widehat{l'} \mapsto \left\langle \widehat{l}, \iota, \mathcal{Q}\right\rangle\right]}{mirror_{Indy}\left(l, \widehat{l}, \sigma, \varsigma\right) = \left(\widehat{l'}, \sigma', \varsigma'\right)}$$

Figure 5.26 Mirroring of contracts.

A property assignment to a sandboxed object is not allowed and immediately signals a sandbox violation. Sandbox violations should not be confused with contract violations as the sandbox violation is neither the subject's nor the context's fault. Technically, it can be seen as a subject violation of the current flat contract or contract constructor as it compares to a programming error in the contract.

5.5 Monitoring Semantics

TreatJS provides three general monitoring semantics: *Lax*, *Picky*, and *Indy*. All three semantics are derived from existing monitoring semantics in the literature [41, 10, 26].

In general, contract monitoring should not interfere with the contract abiding execution of a host program. That is, as long as the host program does not violate any contract the program execution should not be influenced by the insertion of a contract. For example, one prominent question is whether the domain contract on an argument value should be present while evaluating the range contract of a (dependent) function contract. To demonstrate this, let's consider function $\lambda f.f$ and contract definition $\mathcal{E}$, where

$$\mathcal{E} = ((Positive \to Positive) \to Test)$$

and $Positive = \texttt{flat}\left(\lambda x.\left(\texttt{>}\, x 0\right)\right)$ and $Test = \texttt{flat}\left(\lambda f.\left(\texttt{=}\left(\left(f\, 0\right)\right) 0\right)\right)$. Moreover, let's apply $\left(\lambda f.f\right) @^{\ell} \mathcal{E}$ to $\lambda x.x$. Even though $\lambda x.x$ satisfies both contracts, $Positive \to Positive$ and $Test$, the range contract may violate the function contract on its subject value. More precisely, a possible violation depends on the visibility of the contract inside $Test$ predicate. Section 4.6 and Chapter 19 contain further examples and a more detailed discussion about this.

TreatJS reorganizes all contract monitors on values that flow into the sandbox of another contract (predicate) according to the chosen monitoring semantics. In contrast to the compatibility check (cf. Section 5.6), which drops contracts that should not be visible during the evaluation of a predicate and which only applies to argument values of a function with a function contract, the reorganization is implemented in the sandbox membrane and applies to every external value used inside of a predicate.

Reorganizing contracts is implemented through the meta-functions $mirror_{Lax}\left(l, \widehat{l}, \sigma, \varsigma\right)$, $mirror_{Picky}\left(l, \widehat{l}, \sigma, \varsigma\right)$, and $mirror_{Indy}\left(l, \widehat{l}, \sigma, \varsigma\right)$, all of which mirror contracts from a source

$$\textbf{\textit{Constraint}} \quad \ni \quad \varphi \quad += \quad \flat \ltimes \flat' \blacktriangleleft \iota \quad \textit{(fork constraint)}$$

Figure 5.27 Syntax extension of constraints.

object l to a sandbox internal object $\widehat{l}$. The functions are called on each $\mathit{wrap}\,(b, \rho, \sigma, \varsigma)$ operation in a sandbox. Figure 5.26 shows the definition of the three functions.

The *Lax* meta-function (Rule Lax) ignores all contract monitors on l and simply returns $\widehat{l}$ without any contract. Removing all contract monitors is correct as it guarantees that a well-behaved program never gets blamed for a contract violation that happens when evaluating predicate code. However, it is not complete as contract violations on values remain unobserved inside of predicate code.

The *Picky* meta-function (Rule Picky) transfers the contract on l to $\widehat{l}$. This guarantees that all contracts remain active when evaluating predicate code. However, this might wrongly blame the context or the subject for violations that happen while evaluating predicate code.

The *Indy* meta-function (Rule Indy) first creates a new blame variable ι_1 and a new blame label ℓ which is associated with the current context κ before it asserts the contract in respect to the new variable. A new *fork constraint* $\flat \ltimes \ell \blacktriangleleft \iota_1$ (Figure 5.27) links the outcome of ι_1 to ι **and** ℓ, whereby subject violations on ι_1 are linked to ι and context violations on ι_1 are linked to ℓ. This guarantees that wrong uses (context violations) of $\widehat{l}$ inside of a predicate will blame the predicate for violating the contract, whereas observed subject violations are still linked to the original constraint set.

Technically, none of the terms mirrors the entire chain potentially nested contract wrappers. Each evaluation copies only one contract on top of l. This is because the recursive walk-through is already handled by the $\mathit{wrap}\,(b, \rho, \sigma, \varsigma)$ operation, as shown in Figure 5.23.

5.6 Compatibility

Compatibility of contracts is needed as the sequential assertion of intersection and union contracts (cf. 5.4.4) mixes up contracts from different sides of an intersection. The issue arises when contract monitors remain enabled during the evaluation of predicate code. To illustrate the need for compatibility let's consider function $\lambda f.f$ and contract definition $\mathcal{E}$, where

$$\mathcal{E} = ((\mathit{Positive} \to \mathit{Positive}) \to \mathit{TestP}) \cap ((\mathit{Negative} \to \mathit{Negative}) \to \mathit{TestN})$$

and $\mathit{Positive} = \texttt{flat}\,(\lambda x.\,(\texttt{>}\,x\,0))$, $\mathit{Negative} = \texttt{flat}\,(\lambda x.\,(\texttt{<}\,x\,0))$, $\mathit{TestP} = \texttt{flat}\,(\lambda f.\,(\texttt{>}\,(f\,1)\,0))$, and $\mathit{TestN} = \texttt{flat}\,(\lambda f.\,(\texttt{<}\,(f\,{-1})\,0))$. Obviously, $\lambda f.f$ satisfies both intersected contracts, $(\mathit{Positive} \to \mathit{Positive}) \to \mathit{TestP}$ **and** $(\mathit{Negative} \to \mathit{Negative}) \to \mathit{TestN}$. But if we apply $(\lambda f.f)\ @^{\ell}\mathcal{E}$ to $\lambda x.x$ and ignore the compatibility side condition, we may end up in a contract violation[1].

This is because the argument value, here function $\lambda x.x$, gets contracted with both domain contracts, $\mathit{Positive} \to \mathit{Positive}$ and $\mathit{Negative} \to \mathit{Negative}$. As both domain contracts are function contracts (delayed contracts), they stay with the argument value until the value is used in an application. If both contract monitors remain active when passing the argument value to one of the predicates on the range, then each range contract violates

[1] More precisely, the result depends on the chosen monitoring semantics (cf. Section 5.5)

the domain contract of the other function contract. However, *Positive* → *Positive* and *Negative* → *Negative* do not belong to the same operand of the intersection and therefore *Negative* → *Negative* must not be enforced in the body of *TestP*, and vice versa. But, this only applies to predicates that belong to the same top-level contract and should not be confused with the application of different monitoring semantics (cf. Section 5.5).

The solution is to *drop* all delayed contracts that belong to a different operand of an enclosing intersection or union contract. We can determine this mismatch by recognizing that the blame variable associated with *Negative* → *Negative* belongs to the right side of the intersection, whereas the blame variable of *TestP* belongs to the left side. So, the predicate evaluation must check the *compatibility* of all delayed contracts on a subject value before passing the value to the predicate. Non-compatible delayed contracts shall be removed.

To define compatibility, we first need to determine to which top-level assertion and to which operand of an intersection or union a contract component belongs. To do so, we compute the set of all possible paths from source blame labels ℓ to the blame variables ι of the current contract. Depending on the chosen evaluation semantics, this may either be a single path or a set of paths in case of the Indy semantics.

Each step in a path is drawn from a set $\boldsymbol{Step} ::= \{\circ, \downarrow, \|\} \times \iota$, where $\circ$ stands for a root label, $\downarrow$ for an indirection in a function, object, or flat contract, and $\|$ for a parallel observation of an intersection or union contract. Furthermore, we define a *path* $\pi \in \boldsymbol{Step}^+$ as a finite sequence of steps.

▶ **Definition 1** (Path). Let $\pi \in \Omega(\flat, \varsigma)$ be a path from $\flat$ to a blame label ℓ in ς. The set of paths $\Omega(\flat, \varsigma) \subseteq \boldsymbol{Step}^+$ is inductively defined by:

$$\Omega(\flat,\varsigma) = \begin{cases} \{(\circ,\ell)\}, & \flat = \ell \\ \{\pi.(\downarrow,\iota) \mid \pi \in \Omega(\flat',\varsigma)\}, & \flat' \blacktriangleleft \iota \in \varsigma, \flat = \iota \\ \{\pi.(\downarrow,\iota) \mid \pi \in \Omega(\flat',\varsigma)\}, & \flat' \blacktriangleleft \neg\iota \in \varsigma, \flat = \iota \\ \{\pi.(\downarrow,\iota_i) \mid \pi \in \Omega(\flat',\varsigma)\}, & \flat' \blacktriangleleft \iota_1 \to \iota_2 \in \varsigma, \flat = \iota_i, i \in \{1,2\} \\ \{\pi.(\|,\iota_i) \mid \pi \in \Omega(\flat',\varsigma)\}, & \flat' \blacktriangleleft \iota_1 \cap \iota_2 \in \varsigma, \flat = \iota_i, i \in \{1,2\} \\ \{\pi.(\|,\iota_i) \mid \pi \in \Omega(\flat',\varsigma)\}, & \flat' \blacktriangleleft \iota_1 \cup \iota_2 \in \varsigma, \flat = \iota_i, i \in \{1,2\} \\ \{\pi.(\|,\iota) \mid \pi \in \Omega(\flat_1,\varsigma) \cup \Omega(\flat_2,\varsigma)\}, & \flat_1 \ltimes \flat_2 \blacktriangleleft \iota \in \varsigma, \flat = \iota \end{cases}$$

To demonstrate paths, let ς be the constraint that arises during the evaluation of $\left((\lambda f.f)\ @^{\ell}\mathcal{E}\right)\ \lambda x.x$, where

$$\varsigma = \cdots : \iota_1 \blacktriangleleft \iota_5 \to \iota_6 : \iota_2 \blacktriangleleft \iota_3 \to \iota_4 : \iota_0 \blacktriangleleft \iota_1 \cap \iota_2 : \ell \blacktriangleleft \iota_0 : \cdot$$

and ι_3 is associated with *Negative* → *Negative*, ι_5 with *Positive* → *Positive*, ι_4 with *TestN*, and ι_6 with *TestP*. The order of constraints seems to be confusing at first, but it corresponds to the evaluation order of contracts. The path sets of ι_3, ι_5 and ι_6 look as follows:

$$\Omega(\iota_3,\varsigma) = \{(\circ,\ell).(\downarrow,\iota_0).(\|,\iota_2).(\downarrow,\iota_3)\}$$

$$\Omega(\iota_5,\varsigma) = \{(\circ,\ell).(\downarrow,\iota_0).(\|,\iota_1).(\downarrow,\iota_5)\}$$

$$\Omega(\iota_6,\varsigma) = \{(\circ,\ell).(\downarrow,\iota_0).(\|,\iota_1).(\downarrow,\iota_6)\}$$

The paths clearly indicate that both blame variables ι_3 and ι_6 belong to different operands of an intersection of the same top-level contract, whereas ι_5 and ι_6 belong to the same operand. It remains to define compatibility of paths.

$$comp\left(\pi, \epsilon\right) \qquad comp\left(\epsilon, \pi\right) \qquad \frac{\ell_1 \neq \ell_2}{comp\left(\left(\circ, \ell_1\right).\pi_2, \left(\circ, \ell_2\right).\pi_2\right)} \qquad \frac{comp\left(\pi_1, \pi_2\right)}{comp\left(\left(\circ, \ell\right).\pi_1, \left(\circ, \ell\right).\pi_2\right)}$$

$$\frac{\flat_1 \neq \flat_2}{comp\left(\left(\downarrow, \flat_1\right).\pi_1, \left(\downarrow, \flat_2\right).\pi_2\right)} \qquad \frac{comp\left(\pi_1, \pi_2\right)}{comp\left(\left(\downarrow, \flat\right).\pi_1, \left(\downarrow, \flat\right).\pi_2\right)} \qquad \frac{comp\left(\pi_1, \pi_2\right)}{comp\left(\left(\|, \flat\right).\pi_1, \left(\|, \flat\right).\pi_2\right)}$$

Figure 5.28 Compatibility of paths.

$$\textsc{Drop-Constant}\quad drop\left(c, \flat, \sigma, \varsigma\right) = \left(c, \sigma\right)$$

$$\textsc{Drop-NoContract}\quad \frac{\langle w, \iota, \mathcal{Q}\rangle \neq \sigma\left(l\right)}{drop\left(l, \flat, \sigma, \varsigma\right) = \left(l, \sigma\right)}$$

$$\textsc{Drop-Compatible}\quad \frac{\langle w, \iota, \mathcal{Q}\rangle = \sigma\left(l\right) \qquad comp_{\varsigma}\left(\flat, \iota\right) \qquad drop\left(w, \flat, \sigma, \varsigma\right) = \left(\sigma', l'\right) \qquad l'' \notin dom\left(\sigma\right) \qquad \sigma'' = \sigma\left[l'' \mapsto \langle l', \iota, \mathcal{Q}\rangle\right]}{drop\left(l, \flat, \sigma, \varsigma\right) = \left(l'', \sigma''\right)}$$

$$\textsc{Drop-Incompatible}\quad \frac{\langle w, \iota, \mathcal{Q}\rangle = \sigma\left(l\right) \qquad \neg comp_{\varsigma}\left(\flat, \iota\right) \qquad drop\left(w, \flat, \sigma, \varsigma\right) = \left(l'\sigma'\right)}{drop\left(l, \flat, \sigma, \varsigma\right) = \left(l', \sigma'\right)}$$

Figure 5.29 Drop delayed contracts.

▶ **Definition 2** (Compatibility of Paths)**.** Two paths π_0,$\pi_1 \in \boldsymbol{Step}^*$ are compatible (written $comp\left(\pi_0, \pi_1\right)$) if one is a prefix of the other or if they have a common prefix and proceed on a different indirections. Figure 5.28 shows the inductive definition of $comp\left(\pi_1, \pi_2\right)$.

▶ **Definition 3** (Compatibility of Blame Identifiers)**.** Two blame identifiers $\flat_0$,$\flat_1$ are compatible w.r.t. constraint list ς (written $comp_{\varsigma}\left(\flat_0, \flat_1\right)$) if the are compatible on all paths in ς:

$$comp_{\varsigma}\left(\flat_0, \flat_1\right) = \forall \pi_0 \in \Omega\left(\flat_0, \varsigma\right), \pi_1 \in \Omega\left(\flat_1, \varsigma\right).comp\left(\pi_0, \pi_1\right)$$

Paths that arise from different blame labels ℓ or that proceed on different indirections are always compatible. Two paths with a common prefix are compatible if they proceed on different indirection, i.e. their first difference must not be an intersection or union. Different instantiations of the same contract are always compatible and can interact with each other. Similarly, the domain and the range contract of a function contract or the properties of an object contract can interact arbitrarily.

To drop non-compatible contracts the metafunction $drop\left(v, \flat, \sigma, \varsigma\right)$ recursively discards all delayed contract monitors on l whose blame variable ι is not compatible with blame identifier $\flat$. Figure 5.29 shows the evaluation rules for dropping non-compatible delayed contracts.

Rule Drop-NoContract returns target value l if no contract is available. Technically we know that incompatible contracts of the same assertion are also on top. They are never nested inside of a sandbox proxy so we can stop dropping if we find an object that is not a contract proxy. Rule Drop-Compatible recursively proceeds with the compatibility check on the proxies target value and re-asserts a compatible contract to the resulting behavior. Rule Drop-Incompatible, in contract, drops an incompatible contract and returns the behavior from the recursive lookup on the proxies target value.

$$\text{CS-Empty}\quad \mu \models \cdot \qquad\qquad \text{CS-Extension}\quad \frac{\mu \models \varphi \qquad \mu \models \varsigma}{\mu \models \varphi : \varsigma}$$

Figure 5.30 Constraint list satisfaction.

Notes

Compatibility of paths should not be confused with the implementation of different monitoring semantics which also reorganizes contract monitors on values passed to predicate functions. Compatibility is more general and handles the visibility of contracts inside of predicate code. However, when using the *Lax* monitoring semantics, which always removes all contract monitors on values that were passed to a predicate, no further compatibility check is required. In all other cases, the compatibility check is strictly required to guarantee symmetry of intersection and union contracts.

5.7 Blame Calculation

The dynamics in Figure 5.19, 5.20, 5.21, and 5.22 use constraints to create a structure for computing contract violations according to the semantics of subject and context satisfaction, respectively. To this end, each blame identifier $\flat$ is associated with a record that defines the blame assignment for the contract related to $\flat$. The record contains two boolean fields, $\flat.\mathit{context}$ and $\flat.\mathit{subject}$. Intuitively, if $\flat.\mathit{context}$ is false, then there is a context that does not respect contract $\flat$ and may lead to negative blame for $\flat$. If $\flat.\mathit{subject}$ is false, then the contract associated with $\flat$ is not subject-satisfied and may lead to positive blame for $\flat$.

5.7.1 Constraint Satisfaction

Computing a blame assignment boils down to computing an interpretation for a constraint list ς. An *interpretation* μ of a constraint list ς is a mapping from blame identifiers $\flat$ to records of elements of $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, such that all constraints are satisfied.

We order truth values by $\texttt{true} \sqsubset \texttt{false}$ and write $\sqsubseteq$ for the reflexive closure of that ordering. This ordering reflects the gathering of information with each execution step and models the accumulated knowledge about the truth of a proposition. Contracts are counted as `true` before its evaluation and `false` signals an observed violation of that proposition.

Formally, we specify the mapping by function

$$\mu \in (\!|\flat|\!) \times \{\mathit{context}, \mathit{subject}\}) \to \mathbb{B}$$

where $(\!|\flat|\!)$ ranges over blame identifiers $\flat$. Constraint list satisfaction and constraint satisfaction are specified by a relation $\mu \models \varsigma$ and $\mu \models \varphi$. Figure 5.30 and Figure 5.31 specify the relation. The rules apply a constraint mapping μ to boolean expressions over constraint variables. This application stands for the obvious homomorphic extension of the mapping.

Every mapping satisfies the empty constraint list $\cdot$ (CS-Empty). The extension of a constraint list ς with a constraint φ corresponds to the intersection of possible solutions on both, φ and ς' (CS-Extension).

The indirection constraint (rule CT-Indirection) just links to another blame identifier. Rule CT-Flat sets the subject satisfaction to the boolean interpretation $\tau(v)$ of the outcome

$$\textsc{CT-Indirection}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota.\mathit{subject}) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota.\mathit{context})}{\mu \models \ell \blacktriangleleft \iota}$$

$$\textsc{CT-Flat}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \tau(v) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \texttt{true}}{\mu \models \flat \blacktriangleleft v}$$

$$\textsc{CT-Inversion}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota.\mathit{context}) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota.\mathit{subject})}{\mu \models \flat \blacktriangleleft \neg\iota}$$

$$\textsc{CT-Function}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{context} \wedge (\iota_1.\mathit{subject} \Rightarrow \iota_2.\mathit{subject})) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{subject} \wedge \iota_2.\mathit{context})}{\mu \models \flat \blacktriangleleft \iota_1 \to \iota_2}$$

$$\textsc{CT-Intersection}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{subject} \wedge \iota_2.\mathit{subject}) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{context} \vee \iota_2.\mathit{context})}{\mu \models \flat \blacktriangleleft \iota_1 \cap \iota_2}$$

$$\textsc{CT-Union}\quad \frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{subject} \vee \iota_2.\mathit{subject}) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{context} \wedge \iota_2.\mathit{context})}{\mu \models \flat \blacktriangleleft \iota_1 \cup \iota_2}$$

$$\textsc{CT-Fork}\quad \frac{\mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota.\mathit{context}) \qquad \mu(\flat.\mathit{subject}) \sqsupseteq \texttt{true} \qquad \mu(\flat'.\mathit{context}) \sqsupseteq \texttt{true} \qquad \mu(\flat'.\mathit{subject}) \sqsupseteq \mu(\iota.\mathit{subject})}{\mu \models \flat \ltimes \flat' \blacktriangleleft \iota}$$

Figure 5.31 Constraint Satisfaction.

b of a predicate. Meta-function $\tau : (\!|v|\!) \to \mathbb{B}$ (Figure 5.32) translates the $\lambda_{\mathcal{CON}}$ values to truth values $\mathbb{B}$ by stripping delayed contracts and interpreting the outcome according to JavaScript's definition of truthy and falsy. A flat contract never blames its context so that $\flat.\mathit{context}$ is always true.

Rule CT-Inversion determines the blame assignment for an object contract. Initially, the subject of an object contract is responsible for returning property values according to their specification, whereas the context is responsible for using all properties according to its specification. This does not require a special constraint. However, after writing a property, the responsibility for the value has changed. Now, the context is responsible for the value that is later returned from the contracted object.

Rule CT-Function determines the blame assignment for a function contract $\flat$ from the blame assignment for the argument and result contracts, which are available through ι_1 and ι_2. Let's first consider the subject part. A function satisfies contract $\flat$ if it satisfies its obligations towards its argument $\iota_1.\mathit{context}$ **and** if the argument satisfies its contract then the result satisfies its contract. The first part arises if the function is a higher-order function, which may pass illegal arguments to its function-arguments. The second part is partial

$$\tau(v) = \begin{cases} \texttt{false} & v \in \{\texttt{false}, 0, \text{""}, \texttt{null}, \texttt{undefined}\} \\ \texttt{true} & \text{otherwise} \end{cases}$$

Figure 5.32 Mapping values to truth values.

correctness of the function with respect to its contract. A function's context (caller) satisfies the contract if it passes an argument that satisfies contract $\iota_1.\mathit{subject}$ **and** uses the result according to its contract $\iota_2.\mathit{context}$. The second part becomes non-trivial with functions that return functions.

Rule CT-Intersection determines the blame assignment for an intersection contract at $\flat$ from its constituents at ι_1 and ι_2. A subject satisfies an intersection contract if it satisfies both constituent contracts: $\iota_1.\mathit{subject} \wedge \iota_2.\mathit{subject}$. A context, however, has the choice to fulfill one of the constituent contracts: $\iota_1.\mathit{context} \vee \iota_2.\mathit{context}$.

Dually, rule CT-Union determines the blame assignment for a union contract at $\flat$ from its constituents at ι_1 and ι_2. A subject satisfies a union contract if it satisfies one of the constituent contracts: $\iota_1.\mathit{subject} \vee \iota_2.\mathit{subject}$. A context, however, needs to fulfill both constituent contracts: $\iota_1.\mathit{context} \wedge \iota_2.\mathit{context}$, because it does not know which contract is satisfied by the subject.

Finally, rule CT-Fork separates subject and context responsibility for a contract at $\flat$ **and** $\flat'$, whereas $\flat$ indicates the initial assertion of the contract and $\flat'$ indicates the context in which the contracted value is used. A subject must satisfy its contract in all contexts of use. However, the context at $\flat$ is not responsible for uses at $\flat'$. Therefore, $\flat.\mathit{context}$ and $\flat'.\mathit{subject}$ are always true.

5.7.2 Solving Constraints

To determine whether a constraint list ς is a blame state (i.e., whether it should signal a contract violation), we check if the constraint list ς maps any source-level blame label ℓ to `false`. To this end, we define the semantics of a constraint list $[\![\varsigma]\!]$ as the least solution that can be computed from ς.

▶ **Definition 4** (Least Solution)**.** The semantics $[\![\varsigma]\!] \in (\!|\flat|\!| \times \{\mathit{context}, \mathit{subject}\}) \to \mathbb{B}$ of a constraint list ς is the least solution that can be computed for ς. That is, $[\![\varsigma]\!] \models \varsigma$ and $[\![\varsigma]\!] \subseteq \mu$ for all $\mu \models \varsigma$.

▶ **Definition 5** (Blame State)**.** ς is a *blame state for blame label* ℓ iff

$$\exists \ell. [\![\varsigma]\!](\ell.\mathit{subject}) \wedge [\![\varsigma]\!](\ell.\mathit{context}) \sqsupseteq \texttt{false}.$$

ς is a *blame state* if there exists a blame label ℓ such that ς is a blame state for this label.

5.7.3 Introducing Blame

Figure 5.33 shows the evaluation rules for blame calculation in $\lambda_{\mathcal{CON}}$. Blame calculation starts immediately after evaluating a predicate. The term $v@^{\iota}w$ checks the outcome w of evaluating a predicate on v. All three rules update the constraint list with a new constraint that maps the outcome of the predicate to blame variable ι before they check if the updated constraint list in a blame state.

$$\textsc{Unit}\quad \frac{\varsigma' = \iota \blacktriangleleft w : \varsigma \qquad \nexists \ell.[\![\varsigma']\!]\,(\ell.\mathit{context}) \sqsupseteq \texttt{false} \qquad \nexists \ell.[\![\varsigma']\!]\,(\ell.\mathit{subject}) \sqsupseteq \texttt{false}}{\kappa, \rho \vdash \sigma, \varsigma, v@^{\iota} w \Downarrow \sigma, \varsigma', v}$$

$$\textsc{Context-Blame}\quad \frac{\varsigma' = \iota \blacktriangleleft w : \varsigma \qquad \exists \ell.[\![\varsigma']\!]\,(\ell.\mathit{context}) \sqsupseteq \texttt{false}}{\kappa, \rho \vdash \sigma, \varsigma, v@^{\iota} w \Downarrow \sigma, \varsigma', \texttt{-blame}^{\ell}}$$

$$\textsc{Subject-Blame}\quad \frac{\varsigma' = \iota \blacktriangleleft w : \varsigma \qquad \exists \ell.[\![\varsigma']\!]\,(\ell.\mathit{subject}) \sqsupseteq \texttt{false}}{\kappa, \rho \vdash \sigma, \varsigma, v@^{\iota} w \Downarrow \sigma, \varsigma', \texttt{+blame}^{\ell}}$$

Figure 5.33 Evaluation rules for blame calculation.

If there exists no blame label ℓ with $\ell.\mathit{context} \sqsupseteq \texttt{false}$ or $\ell.\mathit{subject} \sqsupseteq \texttt{false}$ (Rule Unit) then the evaluation returns the subject value and the updated constraint list. If $\ell.\mathit{context} \sqsupseteq \texttt{false}$ (Rule Context-Blame) then ς' is a blame state for label ℓ. Consequently, the evaluation results in contract violation $\texttt{-blame}^{\ell}$ blaming the context for violating the contract related to ℓ. Otherwise, if $\ell.\mathit{subject} \sqsupseteq \texttt{false}$ (Rule Subject-Blame) then ς' is a blame state for label ℓ resulting in a contract violation $\texttt{+blame}^{\ell}$ blaming the subject for violating the contract related to ℓ.

5.7.4 Constraint Graphs

To demonstrate blame calculation with constraints let's consider function $\mathit{addOne} = \lambda x.\,(\texttt{+}\,x\,1)$ and function contract $\mathcal{E}$ where

$$\mathcal{E} = (\mathit{Even} \to \mathit{Even}) \cap (\mathit{Positive} \to \mathit{Positive})$$

and $\mathit{Even} = \texttt{flat}\,(\lambda x.\,(\texttt{=}\,(\texttt{mod}\,x\,2)\,0))$. Figure 5.34 shows the constraint graph after applying *addOne* to the number value 1 and after applying *addOne* to the number value 2. Obviously, *addOne* does not satisfy $\mathcal{E}$, but as contract monitoring of higher-order functions is based on testing we see this violation only in a particular situation.

In a first call, we apply *addOne* to the number value `1`. While doing this, the left function contract $(\mathit{Even} \to \mathit{Even})$ fails, blaming the context of that function for not calling *addOne* with an even number, whereas the right function contract $(\mathit{Positive} \to \mathit{Positive})$ succeeds. Because the context of an intersection can choose to use the *addOne* either with positive or even numbers, the intersection contract is satisfied. In the second call, we apply *addOne* to the number value 2, which raises a contract violation blaming the subject (*addOne*) of the intersection. In this situation, the first function contract fails blaming the subject for not return an even number, whereas the right function contract succeeds. As the subject of an intersection needs to fulfill both sides, the intersection contract fails, blaming the subject.

In our second experiment, we demonstrate the differences between *Lax*, *Picky*, and *Indy* monitoring semantics. To this end, we consider function $\lambda f.f$ and contract $\mathcal{F}$ with

$$\mathcal{F} = (\mathit{Positive} \to \mathit{Positive}) \to \mathit{Test}$$

and $\mathit{Positive} = \texttt{flat}\,(\lambda x.\,(\texttt{>}\,x0))$ and $\mathit{Test} = \texttt{flat}\,(\lambda f.\,(\texttt{=}\,((f\,0))\,0))$. In all three experiments, we apply $\lambda f.f$ to $\lambda x.x$. During the evaluation, the domain contract (here $\mathit{Positive} \to \mathit{Positive}$)

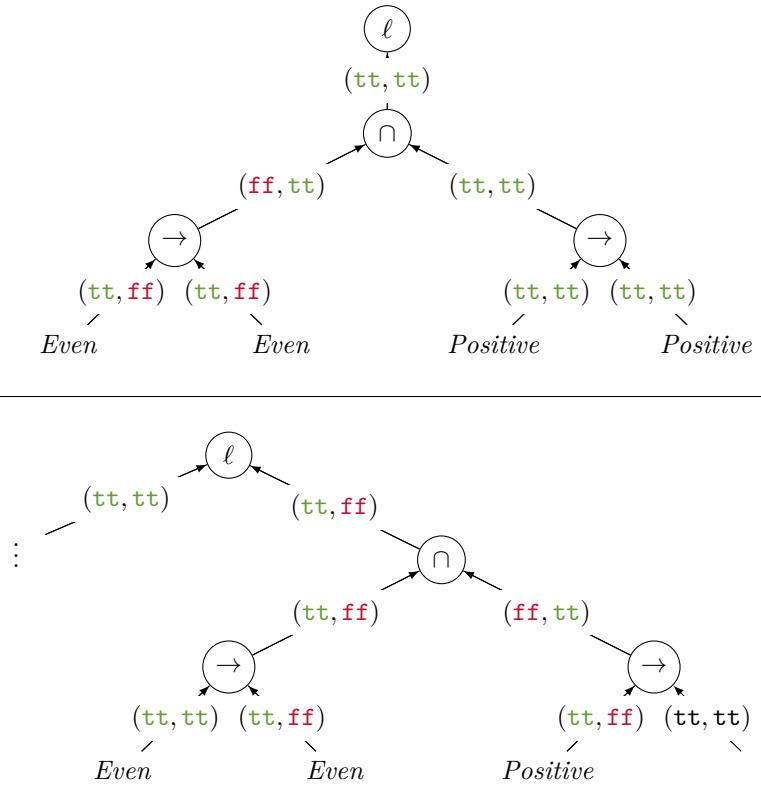


Figure 5.34 Blame calculation of function $addOne = \lambda x.\,(+\,x\,1)$ contracted with $(Even \to Even) \cap (Positive \to Positive)$. The top shows the constraint graph after applying `addOne` to the number value 1 (first call). The bottom shows the extended graph after applying *addOne* to the number value 2 (second call). Each node is a constraint and each edge references to a blame variable. The labeling shows the values of the associated $\{context, subject\}$ record.

gets applied to argument ($\lambda x.x$) before the base contract on the range is applied to the function's return. Obviously, $\lambda x.x$ satisfies the domain contract $Positive \to Positive$ but *Test* violates the function contract on its subject value, and thus it may violate the contract on $\lambda x.x$.

Figure 5.35 shows the constraint graph that arises when using the *Lax* monitoring semantics. The *Lax* monitor removes all contracts from $\lambda x.x$ before it uses the function as a subject value. So, the function contract is not tested and no contact violation is reported. A quick recap: undefined blame variable are assumed to be *true*, more precisely as $\{\texttt{tt}, \texttt{tt}\}$.

Figure 5.36 shows the constraint graph that arises when using the *Picky* monitoring semantics. In *Picky*, contract monitors stay on all values that were passed to predicate function. So, evaluating the predicate violates the function contract on $\lambda x.x$ because it applies subject value to a non-positive number value. Thus, the function contract $Positive \to Positive$ reports a context violation. However, as $\lambda f.f$ is responsible for using the argument value according to its specification, the outer function contract reports a subject violation.

Finally, Figure 5.37 shows the constraint graph that arises when using the *Indy* monitoring semantics. *Indy* monitoring reorganizes all contracts on values that are passed to a predicate function. As before, the function contract on $\lambda x.x$ remains active and reports a context violation because the predicate applies the subject value to a non-positive number. However, a new fork constraint separates the context and subject responsibility and forwards the

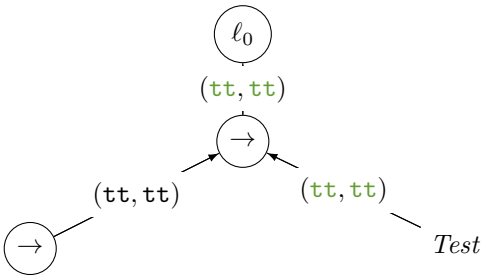


Figure 5.35 Blame calculation using *Lax* monitoring semantics. The figure shows the callback graph after applying function $\lambda x.\,(x\,1)$ contracted with $(\mathit{Positive} \to \mathit{Positive}) \to \mathit{Test}$ to $\lambda x.x$. Each node is a constraint and each edge references to a blame variable. The labeling shows the values of the associated $\{\mathit{context}, \mathit{subject}\}$ record.

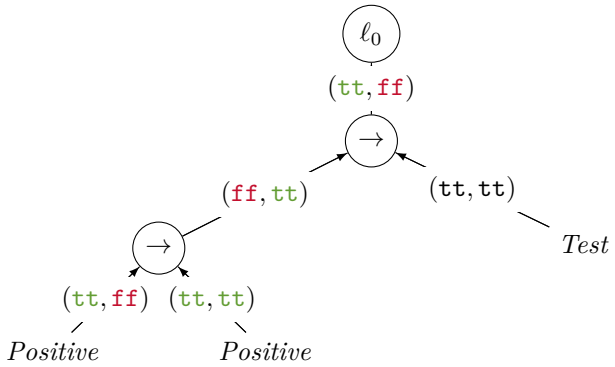


Figure 5.36 Blame calculation using *Picky* monitoring semantics. The figure shows the callback graph after applying function $\lambda x.\,(x\,1)$ contracted with $(\mathit{Positive} \to \mathit{Positive}) \to \mathit{Test}$ to $\lambda x.x$. Each node is a constraint and each edge references to a blame variable. The labeling shows the values of the associated $\{\mathit{context}, \mathit{subject}\}$ record.

contract violation to a new blame label that is associated with the context in *Test*.

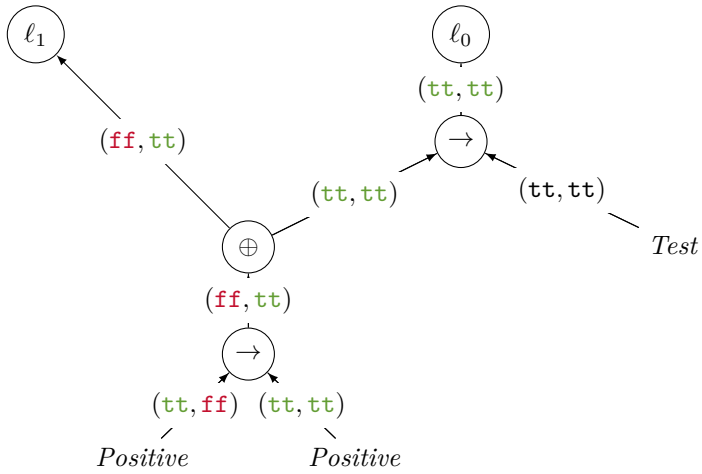


Figure 5.37 Blame calculation using *Indy* monitoring semantics. The figure shows the callback graph after applying function $\lambda x.\,(x\,1)$ contracted with $(\mathit{Positive} \to \mathit{Positive}) \to \mathit{Test}$ to $\lambda x.x$. Each node is a constraint and each edge references to a blame variable. The labeling shows the values of the associated $\{\mathit{context}, \mathit{subject}\}$ record.

6 Implementation

The calculus presented in Chapter 5 is implemented in TreatJS, a language embedded higher-order contract system for full JavaScript. TreatJS is implemented as a library in JavaScript, and all aspects are accessible through a contract API. The library can be deployed as a language extension and does not require changes in the JavaScript run-time system.

The implemented is based on the JavaScript Proxy API [20] which was released with the ECMAScript 6 (ES6) [33] specification in 2015. The API is implemented in most common browsers, i.e., it is implemented in Firefox since version 18.0, in Chrome since version 49.0, and in Edge since version 13. The proxy-based implementation guarantees full interposition for the full JavaScript language and all code regardless of its origin, including dynamically loaded code and code injected via `eval`. The implementation provides full browser compatibility (i.e., all browsers work without modifications as long as they support the JavaScript Proxy API). No source code transformation or avoidance of JavaScript's dynamic features is required.

6.1 Predicates

In TreatJS, predicates are specified by plain JavaScript functions. TreatJS does not impose syntactic restrictions on predicates but expects them to terminate on all inputs. However, predicates should not interfere with the execution of a contract abiding host program. Adding a contract to a program should either result in the same outcome or a contract violation.

6.2 Sandboxing

TreatJS employs sandboxing to keep the predicate evaluation and the evaluation of constructor functions apart from the normal program execution. It uses a limited version of DecentJS (cf. Part II) to guarantee noninterference with the actual program execution.

Like DecentJS, the TreatJS-Sandbox encapsulates argument values in a proxy membrane to enforce write-protection, and it withholds external bindings of functions by recompiling the function inside of the sandbox. Unlike DecentJS, it does not provide effect logging and each attempt to modify a value that is also visible to the host application gives rise to a sandbox violation. Technically it would be possible to use a full blown sandbox like DecentJS, but for efficiency reasons, most of its features remain disabled. More details about the implementation of DecentJS and TreatJS-Sandbox can be found in Chapter 10.

6.3 Constraints

TreatJS uses constraints to link the outcome of a contract to the enclosing contract operators. Constraints are implemented by special JavaScript objects that encompass the actual evaluation state of the constraint along with a callback function that passes the internal state of the constraint to the parent constraint, or at the top level to the assertion. Moreover, each constraint objects provides one callback function per operand which accepts the evaluation state from the associated sub-contracts. The callback function updates the internal state and triggers an update of the parent constraint if the internal state has changed.

6.4 Delayed Contracts

Delayed contracts checking is implemented using JavaScript Proxies [20, 33]. The assertion of a delayed contract wraps the subject value in a proxy object. The handler for the proxy contains the contract and implements a trap function to mediate the corresponding operation on the subject value. It later asserts the contract when the value is used. No source code transformation or change in the JavaScript run-time system is required.

The only special case is the assertion of a delayed contract to a primitive value. As the proxy constructor requires an object as target, the primitive value cannot directly be used as a proxy target. To overcome this limitation, TreatJS uses a dummy object in place of the original subject value. The dummy object overrides the `Symbol.toPrimitive`[1] and the `valueOf` property with a function that returns the primitive value whenever the dummy object is used in an operation that requires a primitive value.

6.5 On Non-interference

The ideal contract system should not interfere with the execution of application code as long as the application does not violate any contract. That is, a contract abiding host program should run as if no contracts where present. In general, we need to distinguish *external* and *internal* non-interference of a contract system.

External Non-interference External non-interference arises from the interaction of the contract system with the host program. Two sources for this are exceptions thrown by the contract system and object equality.

Internal Non-interference Internal non-interference arises from the execution of unrestricted predicate code in a base contract. This code may try to write to data that is visible to the application, it may throw an exception, or it may not terminate.

Exceptions arise when a contract monitor observes a contract violation which is then reported in the form of a contract exception. The host program can catch such an exception and thus the host program become aware of the contract system.

Object equality becomes an issue if contract wrappers are not pointer equal to their wrapped subject values. The problem arises if a subject value with and without a contract is part of the same execution environment. If the wrapper is different (i.e., not pointer-equal) from the wrapped subject value then each equality test between wrapper and subject or between different wrappers for the same subject returns false instead of true.

Our implementation uses JavaScript proxies to implement contract wrappers. Unfortunately, JavaScript proxies are always different from their wrapped target objects, and the only safe way to change that is to modify the underlying JavaScript VM. Part III contains a more detailed discussion of this problem and presents a solution for a transparent contract wrapper.

Internal sources of interference are the execution of unrestricted JavaScript code in the predicate of a base contract or the constructor function of a contract constructor. Plain JavaScript functions implement predicates and constructor functions and use the full expressive power of JavaScript. To guarantee noninterference with the application code,

[1] `Symbol.toPrimitive` is a special JavaScript `Symbol` used to name a function-valued property that converts an object to a corresponding primitive value. The function property is called whenever an object is used in an operation that requires a primitive value.

TreatJS executes predicates and constructor functions in a sandbox that restricts all write operations to local objects and that captures all exceptions thrown by the function (except other contract violations). Unfortunately, a non-terminating execution cannot be interrupted as such a timeout cannot be implemented in JavaScript[2].

6.6 Getting the Source Code

The implementation of TreatJS is available on the Web[3]. It comes along with a PLT Redex model[4] that implements a JavaScript core calculus with contracts and contract monitoring.

[2] Most JavaScript engines like *Node.js* or *Spidermonkey* provide built-in functions to implement timeouts and system interrupts. However, those functions are host extension and not part of the JavaScript standard. Therefore we cannot rely on the presence of such a function.

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

[4] `https://github.com/keil/Contract-Simplification`

7 Runtime Evaluation

This chapter reports on our experience with applying the TreatJS contract system to benchmark programs of the Google Octane 2.0 Benchmark Suite[1]. We primarily focus on the influence of contracts on the execution time of a contract abiding host program.

7.1 Benchmark Programs

Unfortunately, there are no large real-world JavaScript applications with contracts that we could use for benchmarking. So, we had to resort to an automatic generation and insertion of contracts to existing benchmark programs.

As source programs, we use benchmark programs of the Google Octane 2.0 Benchmark Suite. Octane measures a JavaScript engine's performance by running a selection of complex and demanding programs[2] that range from performance tests to real-world web applications, from an OS kernel simulation to a portable PDF viewer. Each program focuses on a specific purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing and compilation, etc.

7.2 The Testing Procedure

To generate contracts for the benchmark programs we wrote a source-to-source compiler that modifies a program by wrapping each function expression[3] in an additional wrapper function. In a first run, this wrapper function is used to install a proxy that, for each call of the function, records the data types of the argument values and the type of the function's return. The recording distinguishes JavaScript's basic data types: `Undefined`, `Null`, `Boolean`, `Number`, `String`, `Symbol`, `Function object`, and `Object`. Afterward, this wrapper function is used to assert a function contract built from the type records to each function expression[4].

Furthermore, if a function's record contains several entries with different types (for example when using a function two times, once with a number and once with a string value), then the wrapper installs an intersection or union contract. However, as some functions return different types for the same input or change their return type for some inputs, not each call confirms with the stated intersection or union contracts. In all such cases, we expect that the contract system throws a contract violation.

Obviously, human developers would state more precise contracts that conform better to the behavior fo that functions. However, as this are automatically generated contracts based on simple type records, finding suitable contracts is not as easy as it seems.

---

[1] `https://developers.google.com/octane`
[2] `https://developers.google.com/octane/benchmark`
[3] A function expression is a function declaration which is part of another expression or statement. In contrast to a function statement, which is always defined in the current function body or in the global scope, function expressions are only defined when the host expression evaluates.
[4] We only consider function expressions as wrapping them did not change the order in which functions are defined.

Benchmark	Indy		Picky		Lax	
	time (sec)	*slowdown*	*time (sec)*	*slowdown*	*time (sec)*	*slowdown*
Richards	19995	4941	18902	4671	18440	4556
DeltaBlue	28431	9910	26327	9176	27146	9462
Crypto	8	1	8	1	8	1
RayTrace	9510	3938	8411	3483	8550	3540
EarleyBoyer	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
RegExp	6	1	6	1	6	1
Splay	162	57	156	55	158	56
SplayLatency	162	57	156	55	158	56
NavierStokes	4	1	4	1	4	1
pdf.js	27	5	25	4	25	4
Mandreel	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
MandreelLatency	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
Gameboy Emulator	1741	464	1596	425	1603	427
Code loading	9	1	9	1	9	1
Box2DWeb	2354	611	2213	574	2172	563
zlib	8	1	8	1	8	1
TypeScript	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	

Figure 7.1 Timings from running the Google Octane 2.0 Benchmark Suite. Column **Indy** gives the execution time and the slowdown for running TreatJS with the Indy monitoring semantics. Column **Picky** shows the time to complete a run with the Picky monitoring semantics, and column **Lax** contains the time required for a run with the Lax monitoring semantics. All measurements were taken from running TreatJS with the Pure safety level. The slowdown is in comparison with Baseline execution time in Figure 7.5.

7.3 Evaluation Results

All benchmark programs were run on a benchmarking machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory. All example runs and measurements were obtained with a modified SpiderMonkey JavaScript engine that treats proxies as transparent. This is required to prevent any interaction between the contract system and the host application.

All runtime measurements were taken from a deterministic run, which requires a predefined number of iterations[5], and by using a warm-up run. Both are predefined configurations in Octane Benchmark Suite. All benchmark programs run between 5 and 8200 times.

Figure 7.1 contains the runtime value and the slowdown factor of all benchmark programs. As expected, all runtime values increase when adding contracts. The benchmark programs run between 1 and 10000 times slower than their baseline, which is listed in Figure 7.5. While some programs like *Richards*, *DeltaBlue*, and *RayTrace* are heavily affected, others are almost unaffected: *Code loading*, *NavierStokes*, *pdf.js*, and *Splay*, for instance.

However, the examples show that the runtime impact of contract assertion depends on the program and on the particular value that is monitored. The impact of a contract strictly depends on the frequency of its application. A contract on a heavily used function (e.g., in *Richards*, *DeltaBlue*, or *RayTrace*) causes a significantly higher runtime deterioration than a contract on a rarely used function. To illustrate this, Figure 7.2 lists some numbers of

[5] Programs run either for one second or for a predefined number of iterations. If there are too few iterations in one second, it runs for another second.

Benchmark	Contract	Assert	Predicate	Membrane	Callback
Richards	24	1599377224	935751200	935751208	935751200
DeltaBlue	54	2320357672	1341331212	1341331220	1341331212
Crypto	1	5	3	11	3
RayTrace	42	687244882	509234422	509234430	509234422
EarleyBoyer	21	201	133	141	136
RegExp	0	0	0	8	0
Splay	10	11624671	7067593	7067601	7067593
SplayLatency	10	11624671	7067593	7067601	7067593
NavierStokes	51	48335	39109	39117	39109
pdf.js	824	1770208	1394694	1394702	1394694
Mandreel	4	14	5	13	8
MandreelLatency	4	14	5	13	8
Gameboy Emulator	3206	141669753	97487985	97487993	97488305
Code loading	5600	34800	18400	18408	18400
Box2DWeb	20075	180252500	112751947	112751955	112820587
zlib	0	0	0	8	0
TypeScript	730	7315	3902	3910	4068

Figure 7.2 Statistic from running the Google Octane 2.0 Benchmark Suite (cont'd). Column **Contract** shows the numbers of top-level contract assertions. Column **Assert** contains the numbers of internal contract assertions whereby column **Predicate** lists the number of predicate evaluations. Column **Membrane** shows the numbers of wrap operations and the last column **Callback** gives the numbers of callback updates.

internal counters. The numbers indicate that the heavily affected benchmarks (*Richards*, *DeltaBlue*, and *RayTrace*) contain a huge number of internal contract assertions. 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 936 million elements, and contract checking performs 936 million callback update operations.

Expressed in absolute time spans, contract checking causes a runtime deterioration of 0.02ms for every single predicate check. For example, the contracted *Richards* requires 19995 seconds to complete and performs 935751200 predicate checks. Its baseline needs 4 seconds. Thus, contract checking requires 19991 seconds. That gives 0.021ms per predicate check.

For better understanding, the number of wrap operations (column **Membrane**) is equal to the number of predicate executions (column **Predicate**) plus eight. This is because we use eight different predicates (each must be wrapped/decompiled once) during the execution of a benchmark program and each predicate accesses only its argument. Moreover, the number of callback updates is also very similar to the number of predicate checks. This is because TreatJS triggers callback updates only if the internal state of a constraint has changed, i.e., we have at least one "initial" callback trigger for each predicate evaluation and one for each failed contract to report the evaluation state to the enclosing contract.

The runtime values in Figure 7.1 also show the difference between the three monitoring semantics, Indy, Picky, and Lax. Overall benchmarks, Indy is the most expensive semantics because it applies a costly contract check on each predicate execution and it increases the number of constraint updates by introducing the third party. The other monitoring semantics have similar runtime impacts.

For sandboxing, the timings in Figure 7.3 shows the runtime difference between the

Benchmark	None		Pure		Strict	
	time (sec)	*ratio*	*time (sec)*	*ratio*	*time (sec)*	*ratio*
Richards	15600	0.78	19995	1.00	19941	1.00
DeltaBlue	22771	0.80	28431	1.00	28959	1.02
Crypto	8	0.97	8	1.00	8	0.97
RayTrace	6123	0.64	9510	1.00	9506	1.00
EarleyBoyer	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
RegExp	6	0.99	6	1.00	6	1.00
Splay	123	0.76	162	1.00	169	1.04
SplayLatency	123	0.76	162	1.00	169	1.04
NavierStokes	4	0.95	4	1.00	4	1.00
pdf.js	20	0.75	27	1.00	27	1.00
Mandreel	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
MandreelLatency	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	
Gameboy Emulator	1302	0.75	1741	1.00	1768	1.02
Code loading	9	0.98	9	1.00	10	1.01
Box2DWeb	1792	0.76	2354	1.00	2406	1.02
zlib	8	1.00	8	1.00	8	1.00
TypeScript	*PositiveBlame*		*PositiveBlame*		*PositiveBlame*	

Figure 7.3 Timings from running the Google Octane 2.0 Benchmark Suite (cont'd). Column **None** gives the execution time and the ratio for running TreatJS without sandboxing. Column **Pure** shows the time to complete a run with the Pure safety level, and column **Strict** contains the time required for a run with the Strict safety level. All measurements were taken from running TreatJS with the Indy monitoring semantics. The ratio is in comparison to the Pure safety level.

different safety levels. The numbers indicate that without sandboxing[6] (column **None**) the benchmarks run approximately 0.16 times faster than with sandboxing (the difference between **None** and **Pure**). The timings also show that Strict sandboxing is about 1.01 times slower than Pure sandboxing (the difference between **Strict** and **Pure**).

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

Benchmark	Full	w/o Predicate		w/o Callback	
	time (sec)	*time (sec)*	*ratio*	*time (sec)*	*ratio*
Richards	19995	13051	6943 (34.73%)	12655	396 (1.98%)
DeltaBlue	28431	18314	10117 (35.58%)	18790	-476 (-1.67%)
Crypto	8	8	0 (2.74%)	8	0 (0.40%)
RayTrace	9510	4601	4909 (51.62%)	4585	17 (0.18%)
EarleyBoyer	-	57	- -	57	- -
RegExp	6	6	0 (0.16%)	6	0 (0.18%)
Splay	162	102	59 (36.77%)	104	-2 (-1.30%)
SplayLatency	162	102	59 (36.77%)	104	-2 (-1.30%)
NavierStokes	4	4	0 (7.43%)	4	0 (-0.44%)
pdf.js	27	17	10 (36.84%)	17	0 (0.94%)
Mandreel	-	5	- -	5	- -
MandreelLatency	-	5	- -	5	- -
Gameboy Emulator	1741	1003	737 (42.35%)	1001	2 (0.12%)
Code loading	9	9	0 (2.55%)	9	0 (-0.19%)
Box2DWeb	2354	1441	914 (38.81%)	1460	-19 (-0.80%)
zlib	8	8	0 (-1.08%)	8	0 (0.59%)
TypeScript	-	2815	- -	2816	- -

Figure 7.4 Timings from running the Google Octane 2.0 Benchmark Suite (cont'd). Column **Full** gives the execution time for running the benchmark without contract monitoring. Column **w/o Predicate** shows the execution time and the ratio of a run without predicate execution, and **w/o Callback** lists the values of a run without predicate execution and without callback updates.

Benchmark	Proxy only		Baseline	
	time (sec)	*ratio*	*time (sec)*	*ratio*
Richards	728	11927 (59.65%)	4	724 (3.62%)
DeltaBlue	1089	17701 (62.26%)	3	1086 (3.82%)
Crypto	8	0 (0.84%)	8	0 (-1.22%)
RayTrace	241	4344 (45.67%)	2	239 (2.51%)
EarleyBoyer	57	- -	56	- -
RegExp	6	0 (-0.51%)	6	0 (0.27%)
Splay	8	96 (59.37%)	3	5 (3.34%)
SplayLatency	8	96 (59.37%)	3	5 (3.34%)
NavierStokes	4	0 (6.01%)	4	0 (0.90%)
pdf.js	7	10 (37.74%)	6	1 (1.96%)
Mandreel	5	- -	5	- -
MandreelLatency	5	- -	5	- -
Gameboy Emulator	53	949 (54.51%)	4	49 (2.80%)
Code loading	9	0 (2.41%)	9	0 (0.77%)
Box2DWeb	96	1364 (57.92%)	4	92 (3.90%)
zlib	8	0 (0.77%)	8	0 (-0.13%)
TypeScript	169	- -	23	- -

Figure 7.5 Timings from running the Google Octane 2.0 Benchmark Suite (cont'd). Column **Proxy only** gives the execution time and the ratio for running the benchmark with a plain forwarding proxy instead of the usual contract proxy. Column **Baseline** gives the baseline execution time and the difference for running TreatJS without contract monitoring.

In addition, Figure 7.4 and Figure 7.5 list the execution time of all benchmark programs in different configurations, which are explained in the figure's caption. The subsequent detailed treatment of the runtime values splits the impact into its individual components.

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 runtime values we find that the predicate execution causes a slowdown of approximately 25% overall benchmarks (the difference between column **Full** and column **w/o Predicate**). The remaining slowdown is caused by the contract system.

Comparing the columns **w/o Predicate** and **w/o Callback** shows that callback updates cause an overall slowdown of -0.1%[7] in our experiment. This point includes the recalculation of the callback constraints as explained in Section 5.7.1. In general, its impact is negligible as TreatJS triggers constraint updates only if the internal state of a constraint has changed.

In the last experiment, we replace the usual contract proxy, which implements delayed contract checking of function and intersection contracts, by a simple forwarding proxy that only forwards the operation to the proxy's target object. This shows the execution time of the programs without predicate execution and without callback updates and splits the overhead of the core contract system and the impact caused by introducing proxy objects. Comparing the columns **w/o Callback** and **Proxy only** indicates that the core contract system (constraint generation, bookkeeping of paths and contexts, introducing proxy objects) decreases the execution time by approximately 34% overall benchmarks.

Finally, column **Baseline** shows the baseline execution time for running the benchmark programs without contract monitoring. The runtime values also indicate that the sole introduction of proxy objects without any functionality causes a slowdown of approximately 2% (difference between column **Proxy only** and column **Baseline**).

For the sake of completeness, Figure 7.6 shows the score values obtained from running the benchmark programs. Google Octane usually reports its results in a score that is inversely proportional to the measured runtime values.

7.4 Assessment

The runtime evaluation in this chapter applies TreatJS to benchmark programs drawn from the Google Octane 2.0 benchmark suite. 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"[8]. Unfortunately, it is pretty hard to measure the real runtime impact of a contract system as there are no real-world applications with contracts that we could use.

To analyze the performance impact of a contract system we either need to add contracts manually or we need to resort to an automatic insertion of contract. Whereas the first way is not representative to serve as a basis for benchmarking, the latter one may end up with contracts in artificial and unnatural places that would be avoided by an efficiency conscious human developer. A contract may repeatedly check the same value or end up on hot paths in a program. Moreover, the automatically inserted contracts come on top of the already existing runtime checks of human developers. JavaScript developers manually test for

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

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

Benchmark *score*	**Baseline** *score*	**Indy** *score*	**Picky** *score*	**Lax** *score*	**None** *score*	**Strict** *score*
Richards	14561	2.9	3.06	3.14	3.71	2.9
DeltaBlue	20899	2.05	2.21	2.14	2.56	2.01
Crypto	13245	12763	13083	13288	13153	13196
RayTrace	38341	9.34	10.6	10.4	14.5	9.34
EarleyBoyer	4491	*PositiveBlame*			*PositiveBlame*	
RegExp	1506	1490	1496	1496	1498	1495
Splay	9142	145	148	148	190	139
SplayLatency	11505	373	379	375	533	325
NavierStokes	15704	14277	14478	14044	14890	14254
pdf.js	9511	1858	1950	1998	2526	1858
Mandreel	11640	*PositiveBlame*			*PositiveBlame*	
MandreelLatency	17616	*PositiveBlame*			*PositiveBlame*	
Gameboy Emulator	32236	60.4	66	65.6	80.7	59.5
Code loading	8192	7876	7857	7877	8077	7720
Box2DWeb	19096	27.7	29.5	30.1	36.3	27.2
zlib	41787	41582	41616	41753	41730	41787
TypeScript	13464	*PositiveBlame*			*PositiveBlame*	

Figure 7.6 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **Baseline** gives the baseline score for running TreatJS without contract monitoring. Column **Indy** contains the scores for running TreatJS with the Indy monitoring semantics. Column **Picky** contains the score values for running TreatJS with the Picky monitoring semantics, and column **Lax** shows the scores from a run with the Lax monitoring semantics. Column **None** shows the scores without sandboxing predicates and Column **Strict** lists the scores with Strict sandboxing.

undefined argument values and check the type of given values by using `typeof` and `instanceof`, all of which become unnecessary when adding a contract that checks the argument.

Thus, the numbers that we obtain give an insight into the performance impact 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.

From the runtime values, we find that the overall slowdown of contract checking vs. a baseline without contracts amounts to a factor of 1508, approximately. The dramatic decrease of the score values in the heavily affected benchmarks is simply caused by the tremendous number of predicate checks that arise during a benchmark run. In relation to this, the runtime impact of approximately 0.02ms per predicate is not very high. Lastly, we have to say that contracts always extend the original program with checks and it is not possible to have dynamic checks at runtime without affecting the execution time. All we can do is to keep the impact as small as possible.

II

Transaction-based Sandboxing for JavaScript

8 Sandboxing JavaScript

JavaScript is used by 94.5%[1] of all the websites. Most of them rely on third-party libraries for feature extensions, to include advertisement, to show maps or calendars, or to connect to a social network. As all these libraries are packed with the application code, the finally executed code is a mix of third-party libraries and scripts from different origins. As not all of these origins are equally trusted, the execution of these scripts should be isolated from one another. However, some scripts must access the application state, and some may be allowed to change it while preserving the confidentiality and integrity constraints of the application.

Unfortunately, the JavaScript language has no built-in namespaces or encapsulation management: there is a global scope for variables, and functions and every loaded script has the same authority. JavaScript developers benefit from JavaScript's flexibility as it enables them to extend the application state easily. On the other hand, once included, a script can access and manipulate every value reachable from the global object.

This part presents the design and implementation of DecentJS, a language-embedded sandbox for full JavaScript which enforces noninterference (integrity and confidentiality) by run-time monitoring. DecentJS enables to run JavaScript functions in a configurable degree of isolation with fine-grained access control. Each sandbox implements a transactional scope, the content of which can be examined, committed, or rolled back. In particular, DecentJS comes with the following features:

Language-embedded sandbox. DecentJS is implemented as a library in JavaScript, and all aspects are accessible through a sandbox API. The library can be deployed as a language extension and does not require changes in the JavaScript run-time system.

Full interposition. The proxy-based implementation guarantees full interposition for the full JavaScript language (ES6 [33]) including all dynamic features (e.g., `with`, `eval`). No source code transformation or prohibition of JavaScript's dynamic features is required.

Shadow Values. Each sandbox implements a fresh scope to run code in isolation to the application state. Proxies implement a membrane [20, 81] and make objects accessible inside of a sandbox. Moreover, they provide effect logging and allow sandbox internal modifications of that object without affecting the original value.

Transaction-based sandboxing. DecentJS provides a transactional scope that logs effects for inspection. Effects reveal conflicts, differences, and changes with respect to another sandbox or the global state. After inspection of the log, effects can be committed to the application state or rolled back.

DecentJS is a spin-off of the language-embedded sandbox used in TreatJS.

8.1 Application Scenarios

This section considers two motivating examples that use DecentJS to guarantee non-interference of JavaScript functions with the host application.

[1] according to http://w3techs.com/, status as of March 2017

8.1.1 TreatJS

TreatJS (Part I) is a higher-order contract system for JavaScript which enforces contracts by run-time monitoring. TreatJS is implemented as a language-embedded system, and all aspects of a contract can be specified using the full JavaScript language.

So, for example, the definition of a base contract `typeNumber` is built from a plain JavaScript function which checks if its argument is a value of type number.

```
let typeNumber = Contract.Base(function (subject) {
  return ((typeof subject) === "number");
}, "typeNumber");
```

Listing 8.1 Construction of a base contract.

In TreatJS, any function can be used as a predicate as any return value can be converted to a boolean value[2]. The assertion of a base contract to a subject value applies the predicate function to that value.

```
Contract.assert(1, typeNumber); // accepted, returns 1
```

Listing 8.2 Assertion of a base contract.

In general, the application of the predicate code should not interfere with the application of the host program. A ground rule for contract monitoring says that the introduction of contracts should not influence a contract abiding host program. However, a predicate may try to write to an object that is visible to the application, it may throw an exception, or it may not terminate.

To guarantee non-interference with the actual program execution, TreatJS evaluates predicates in a sandbox. The sandbox removes all external bindings of the function closure and wraps the argument value in a proxy membrane to prevent the value from unintended modifications. A timeout could be used to stop a non-terminating execution, alas such a timeout cannot be implemented in JavaScript.

More details about sandboxing of contracts may be found in Section 4.5.

8.1.2 Observer Proxies

An observer proxy (cf. Chapter 15) is an extension of the already existing proxy implementation. Its motivation is to restrict proxies to projections: a proxy that either behaves identical to the target object or it restricts the behavior of the target object, for example by throwing an exception or by implementing a membrane. Observer proxies are similar to Racket's chaperone [100] proxies. Such an observer can cause a program to fail more often, but in case it does not fail it will behave in the same way as if no observer is present.

An observer handler provides an API similar to the API of the existing JavaScript proxies. The constructor consumes a `target` object and a `handler`. Like before, the handler is a placeholder object for optional trap functions to "observe" operations on the proxy object. Figure 8.3 shows the implementation of a handler object that throws an exception when reading an undefined property. Otherwise, it continues with the usual operation and returns the property. A detailed explanation of the implementation is given in Section 15.1.

To guarantee noninterference with the actual program code, the observer needs to evaluate all user-defined traps in a sandbox. This ensures that the handler does not manipulate the

[2] JavaScript programmers speak of *truthy* and *falsy* about values that convert to true or false.

```
let handler = {
  get: function(target, name, receiver, continue) {

    // checks for undefined property names
    if(!(name in target))
      throw new Error(‘Access to undefined property ${name}.‘);

    // continue property lookup
    continue(target, name, receiver, function inspect(result, callback) {
      callback(result);
    });
  }
};
```

Listing 8.3 Implementation of a handler object.

proxy's target object and that it does not perform any side effects other than the target object would do. Moreover, the trap might be allowed to cause effects on a certain data structure, for example when updating the evaluation state of a contract. The sandbox can adjust this white-listing.

8.2 Getting the Source Code

The implementation of DecentJS is available on the web[3].

[3] https://github.com/keil/DecentJS

9 Transaction-based Sandboxing: A Primer

Transactional sandboxing is inspired by the idea of transaction processing in database systems [111] and software transactional memory [96]. Each sandbox implements a transactional scope whose internal state cannot be observed from the outside but whose effects can be inspected, committed, or rolled back. This chapter provides a series of examples that explain DecentJS's facilities from a programmer's point of view.

Isolation of code. A DecentJS sandbox can run JavaScript functions in isolation to the application state. Proxies make external values visible inside of the sandbox, provide effect logging, and handle sandbox internal modifications of that value.

Transactional scope. A DecentJS sandbox provides a transactional scope in which effects are logged for inspection. Policy rules can be specified so that only effects that adhere to the rules are committed to the application state, and others are rolled back.

9.1 Cross-sandbox Access

To demonstrate sandboxing, we consider different operations on binary trees, as defined by `Node` and `Leaf` in Listing 9.1. Each node element consists of a value field, a left node, and a right node. A leaf element consists only of a value field. Both prototype objects provide a `toString` method that prints a string of that element. The implementation comes along with some auxiliary functions defined in Listing 9.2. Function `heightOf` computes the height of a node and function `setValue` replaces the value field of a node by its height, recursively.

As a running example, we perform operations on a binary tree consisting of one node and two leaf elements. All value fields are initialized with `0`.

```
1 let root = new Node(0, new Leaf(0), new Leaf(0));
```

Listing 9.3 Definition of a binary tree.

Calling `toString` on `root` returns a sequence of value fields.

```
1 print(root); // prints 0,0,0
```

Listing 9.4 Output of calling `toString`.

Now, instead of calling `setValue` directly, we use DecentJS to avoid unintended modifications on `root`'s data structure and to observe caused effects. To this end, we first create a new sandbox by calling the `Sandbox` constructor. The following code snippet instantiates a fresh sandbox based on the current `this` object, which is the actual global object.

```
1 let sbx = new Sandbox(this, Sandbox.DEFAULT);
```

Listing 9.5 Construction of a new sandbox.

The sandbox constructor takes two arguments: a JavaScript object that acts as the global object of the sandbox and a configuration object containing some preferences. The sandbox object `sbx` now provides two possible uses:

Wrapping values. A value can be wrapped in the sandbox using the `wrap` method of a sandbox object.

Calling function objects. Each sandbox provides a `call`, an `apply`, and a `bind` method with the intuitive meaning known from `Function.prototype`.

```
function Node (value, left, right) {
  this.value = value;
  this.left = left;
  this.right = right;
}
Node.prototype.toString = function () {
  return (this.left + "," + this.value + "," + this.right);
}
function Leaf(value) {
  this.value = value;
}
Leaf.prototype.toString = function () {
  return this.value;
}
```

Listing 9.1 Implementation of a `Node` element.

```
function heightOf (node) {
  if (node instanceof Leaf) return 0;
  else return (Math.max(heightOf(node.left),heightOf(node.right))+1);
}
function setValue (node) {
  if (node instanceof Leaf) node.value=heightOf(node);
  else {
    node.value=heightOf(node);
    setValue(node.left);
    setValue(node.right);
  }
}
```

Listing 9.2 Implementation of some auxiliary functions.

When wrapping a value in a sandbox, a non-function object gets wrapped in a proxy membrane which intercepts all write operations on the target object. A function object gets redefined in our sandbox and wrapped in a special proxy that implements the sandbox membrane on the argument values of that function.

The methods `call`, `apply`, and `bind` evaluate a function object inside of the sandbox. These methods work similar to the methods known from `Function.prototype`, but require a function object as their first argument. Internally, `call`, `apply`, and `bind` apply the `wrap` function to their first argument before they proceed with calling the corresponding prototype method. In the following example, we call the `setValue` inside of sandbox `sbx` to prevent the original `root` node from unintended modifications by calling `setValue`.

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

Listing 9.6 Calling a function in a sandbox.

Internally, the `call` method first decompiles and redefines the `setValue` function in the sandbox before it applies the function to the `root` argument. This step erases all existing bindings to the global scope and builds a new closure with respect to the sandbox environment.

The `this` and `root` value are wrapped in the sandbox membrane before they are forwarded to the function. The "sandbox" proxy forwards reads to the proxy's target object, whereas writes produce a shadow value in the proxy. Subsequent reads on those values will use the sandbox internal value. Clearly, getter and setter functions have to be decompiled as well.

Proxies implement this behavior. Their traps decide which value to use and when to decompile a function. The proxy membrane ensures that all input and return values are also wrapped in the membrane.

An exception with regard to decompiling is the access to a native function, for example when calling `Math.max` in line 3 of Listing 9.2. Native objects must also be wrapped to enforce write protection, but their methods cannot be decompiled because no string representation exists. As recompiling is not possible the only alternative is to trust native functions or to forbid them. Fortunately, most native methods do not produce side effects.

Let's go back to our example. Not surprisingly, the `root` value has not been affected by calling `setValue` in the sandbox.

```
1 print(root); // prints 0,0,0
```

Listing 9.7 Output of calling `toString`.

However, calling `toString` inside of the sandbox shows something different.

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

Listing 9.8 Output of calling `toString` in `sbx`.

9.2 Effects

Beyond access restrictions, each sandbox records the effects on all objects that cross the sandbox membrane. The sandbox distinguishes between *read effects*, *write effects*, and *call effects*. The resulting lists of effects are accessible through `sbx.effects`, `sbx.readeffects`, `sbx.writeeffects`, and `sbx.calleffects` which contain all effects, read effects, write effects, and call effects, respectively. Call effects are distinguished from read and write effects for more detailed conflict detection.

All three lists offer special query methods to select only the effects of a particular object.

```
let effects = sbx.effectsOn(this);
for(effect of effects) print(effect);
```

Listing 9.9 Selecting read effects on `this`.

The code snippet above prints a list of all effects performed on `this`, which is the global object. The list shows read effects to `heightOf`, `Math`, and `Leaf`, as the following extract demonstrates. It contains one entry for each `has` and `get` request during the execution of `setValue`.

```
(#1) get [name=Leaf]
(#1) get [name=Math]
(#1) get [name=heightOf]
(#1) has [name=Leaf]
(#1) has [name=Math]
(#1) has [name=heightOf]
```

Listing 9.10 Read effects on `this`.

The first column shows a unique object identifier, 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 effects to `this`, but there are write effects to `root`.

```
let writeeffects = sbx.writeeffectsOn(root);
for(effect of writeeffects) print(effect);
```

Listing 9.11 Selecting write effects on `root`.

```
(#4) set [name=value]
```

Listing 9.12 Write effects on `root`.

9.3 Inspecting a Sandbox

The sandbox internal state may diverge from the outside state for different reasons. Inspecting the state of a Sandbox distinguish between *differences*, *changes*, and *conflicts*. A *difference* indicates a sandbox-internal change of a value, whereas a *change* indicates a modification of the outside value. A *Conflict* arises when comparing the effects of different sandboxes on the same "outside" value.

9.3.1 Differences

A *difference* reveals a sandbox-internal modification of a value with respect to the corresponding sandbox-external counterpart. Differences can be examined using an API that is very similar to the effect API. There are flags to check whether a sandbox has differences as well as iterators over them.

```
sbx.hasDifferences; // returns true
```

Listing 9.13 Checking for differences.

In our example, `root`'s values field has changed while calling `setValue`. The `differences` list returns a list of all differences.

```
let differences = sbx.differences;
for(difference of differences) print(difference);
```

Listing 9.14 Selecting all differences.

The result of the above query is the following list.

```
Difference: (#4) set [name=value]@SBX001
```

Listing 9.15 Differences in `sbx`.

Differences only fire on values that were written in the sandbox. In all other cases sandbox and global environment return the same value.

9.3.2 Changes

A *change*, the dual of a difference, reveals a sandbox-external modification of a value that is used inside of the sandbox. As before, there exist some query methods.

```
sbx.hasChanges; // returns false
```

Listing 9.16 Checking for changes.

Right now, there are not changes in the global state. To create some, we replace `root`'s left leaf by a new subtree, as shown in the following example.

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

Listing 9.17 Extension of `root`'s left leaf element.

Now, the flag indicates that same outside values has changed.

```
sbx.hasChanges; // returns true
```

Listing 9.18 Checking for changes (cont's).

The list of changes reveals that `root`'s `left` field has been overwritten.

```
let changes = sbx.changes();
for(change in changes) print(change);
```

Listing 9.19 Selection all changes.

```
Change: (#4) get [name=left]@SBX001
```

Listing 9.20 Changes in `sbx`.

To sum up, changes can reveal modifications of the application state with respect to the sandbox-internal value. However, this requires that **1.** the value is accessed once and **2.** the sandbox remembers the value. For example, to recognize that `root`'s `left` field has been changed, the sandbox must remember the "old" value that has been returned when computing the height. So, calling `toString` inside of `sbx` does not show the modifications on `root`.

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

Listing 9.21 Output of calling `toString` in `sbx` after changing `root`'s `left` field.

Once accessed, a sandbox proxy caches the returned value, respectively its wrapped counterpart. This remembering is required to have a consistent view inside of a sandbox. Otherwise, a sandbox would lose all its modifications on a value when changing the access path to that object. Moreover, this caching improves the efficiency as it prevents the target from frequent accesses and the sandbox from frequent wrap operations.

9.3.3 Conflicts

Our sandboxing approach allows us to have multiple sandbox instances, all of which may operate on the same data structure. This may result in multiple identities of the same object which exist in parallel in different sandboxes.

In contrast to differences (cf. Subsection 9.3.1) and changes (cf. Subsection 9.3.2), which reveal differences between a sandbox and the application state, a *conflict* appears between different sandboxes. They are more fine-grained and more efficient because the membranes enables us to observe read and write operations on both sides.

Two sandbox membranes are in *conflict* if at least one sandbox modifies a value that is afterward accessed by the other sandbox. The idea of conflicts corresponds to classical data hazards from parallel executions. Our sandbox distinguishes two types of conflicts:

- *Read-after-Write*
- *Write-after-Write*

A third well-known hazard, *Write-after-Read*, is not handled because the hazard represents a problem with concurrent executions and does not occur in sequential JavaScript. In case a second sandbox writes a value that has been read by the first sandbox, then this write operation is definitely behind the read operation and does not depend on the evaluation order of a scheduler. The first sandbox has to terminate before the second sandbox starts.

To demonstrate conflicts, the following code snippet defines another auxiliary function, `appendRight`, which adds a subtree to the right edge of the given `node` element.

```
1 function appendRight(node) {
2   node.right = new Node('a', new Leaf('b'), new Leaf('c'));
3 }
```

Listing 9.22 Definition of `appendRight`.

Furthermore, let's instantiate another sandbox, `sbx2`.

```
1 var sbx2 = new Sandbox(this, Sandbox.DEFAULT);
```

Listing 9.23 Construction of a second Sandbox.

To recap, the global `root` object prints the string `0,0,0,0,0`, whereas the `root` node in `sbx` prints `0,1,0`. Now, let's evaluate `appendRight` in `sbx2`.

```
1 sbx2.call(appendRight, this, root);
```

Listing 9.24 Calling `appendRight` in a sandbox.

After calling `appendRight`, the `root` node in `sbx2` prints the sequence `0,0,0,0,b,a,c`.

Even though both sandboxes manipulate the same object, they are not in conflict. The following code snippet shows how to test for conflicts.

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

Listing 9.25 Testing for conflicts.

This is because the call of `appendRight` is "behind" the call of `setValue` and because both sandboxes write to different fields: `sbx` recalculates the `value` files, whereas `sbx2` replaces the right edge of it. However, this changes if we call `setValue` again.

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

Listing 9.26 Calling `setValue` in a sandbox.

Now, both sandboxes are in conflict because `sbx2` writes to a field (`right`) that is afterward accessed in `sbx`.

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

Listing 9.27 Testing for conflicts, cont'd.

As before, the sandbox enables a developer to select a list of conflicts.

```
let conflicts = sbx.conflictsWith(sbx2);
for(conflict of conflicts) print(conflict);
```

Listing 9.28 Selecting conflicts between `sbx` and `sbx2`

The call returns the following lines:

```
Conflict: (#4) get [name=right]@SBX001 - (#4) set [name=right]@SBX002
```

Listing 9.29 Conflicts between `abs` and `sbx2`.

The list shows a *read-after-write* conflict, where `sbx` reads the `right` field of an object that has been written (and not committed) by `sbx2`.

9.4 Transaction Processing

After inspecting a sandbox, effects can be committed to the application state or rolled back.

9.4.1 Commits

A sandbox *commit* applies a sandbox-internal modification to the application state. Effects may be committed on a per effect basis, in respect to an object, or all at once by calling `commit` on the sandbox object. To recap, the global `root` object is still *unmodified* and prints the sequence `0,0,0,0,0`. Calling `commit` on `sbx` applies all modification to the global `root` object.

```
sbx.commit();
print(root); // prints 0,0,0,1,0
```

Listing 9.30 Committing effects of a sandbox.

DecentJS enables fine-grained effect and commit-handling. So, a first sandbox can commit its changes on `value` fields, whereas another sandbox can commit changes on edges. However, there is no automatic check of conflicts or changes in front. Committing a value simply overwrites the corresponding target object.

9.4.2 Rollbacks

Borrowing terminology from database systems, a *rollback* specifies an operation which throws existing data manipulations away and returns a value to its previous configuration. Transferred onto our sandbox, a rollback of an effect/object resets the value to its outside counterpart.

For example, consider the `root` node in `sbx2`, which is still the tree that prints the sequence `0,0,0,0,b,a,c`. Like commits, rollbacks are fine-grained. A developer can either rollback a particular effect, all effects of an object, or all effects of a sandbox. Now, let's rollback all modifications in `sbx2`.

```
sbx2.rollback();
```

Listing 9.31 Rollback effects of a sandbox.

Now, all local shadow values are removed and the outside configuration is visible.

9.4.3 Revert

A rollback undoes a particular effect and resets a field to its initial value. However, some effects cannot be rolled back, for example when calling `Object.freeze` on a sandbox proxy. To undo those operations, DecentJS provides a special *revert* operation. A sandbox *revert* removes the entire shadow object of a wrapped value, and thus it undoes all effects at once.

The following code snippet reverts the `root` node in `sbx2`.

```
sbx2.revert();
```

Listing 9.32 Revert a sandbox to the outside state.

Now, all shadow values are removed, and the original value shines through. Moreover, a revert also removes all cached values.

9.5 Pre-state Snapshot

The *snapshot mode* instructs the sandbox to produce local copies of values that were given to the sandbox. The motivation is to have a snapshot of those values and to enable a sandbox to *rebase* to that snapshot.

A snapshot can be generated by giving value to snapshot array, which is the third argument of the sandbox constructor.

```
let sbx3 = new Sandbox(this, Sandbox.DEFAULT, [root]);
```

Listing 9.33 Construction of a new sandbox using the snapshot mode.

The sandbox can be used as usual. For example to call `setValue`:

```
sbx3.call(setValue, this, root);
sbx3.call(root.toString, root); // returns 0,1,0,2,0
```

Listing 9.34 Calling a function in a sandbox with enabled snapshot mode.

Now, to demonstrate the differences, let's again extend `root` with a new subtree on the right.

```
root.right = new Node(0, new Leaf(0), new Leaf(0));
```

Listing 9.35 Extension of `root`'s right leaf element.

Unimpressed by this modification, subsequent `rollback` and `revert` operations return to the initial state (the snapshot), and not to the outside counterpart.

```
sbx3.rollback();
sbx3.call(root.toString, root); // returns 0,1,0,2,0
```

Listing 9.36 Rollback effects of a sandbox with snapshot mode.

However, a special `rebase` method can update the snapshot upon request.

To sum up, the snapshot mode can be used to produce a snapshot states inside of a sandbox and to return to this snapshot. However, the snapshot mode is more expensive than normal shadow values and it should be used with caution.

9.6 Transparent Sandboxing

The *transparent mode* is another special mode of our sandbox. It disables the construction of shadow values inside of a sandbox and applies all modifications directly to the target object. However, it still wraps all values that cross the sandbox boundary in a proxy membrane. Hence, effect logging and all other sandbox features remain available. The transparent mode still enables to investigate the effects of unfamiliar JavaScript code and to check for conflicts between different sandboxes.

The transparent mode can be enabled by using the corresponding sandbox configuration.

```
1 let sbx4 = new Sandbox(this, Sandbox.TRANSPARENT});
```

Listing 9.37 Construction of a new sandbox using the transparent mode.

Like before, we can use `sbx4` to execute JavaScript functions.

```
1 sbx4.call(setValue, this, root);
2 sbx4.call(root.toString, root); // returns 0,1,0,2,0
```

Listing 9.38 Calling a function in a sandbox with enabled transparent mode.

Now, after calling `setValue`, all modifications are immediately committed to the target object. Printing the global `root` element reveals this.

```
1 print(root); // prints 0,1,0,2,0
```

Listing 9.39 Output of calling toString.

9.7 Reverse Sandboxing

This feature allows us to encapsulate sensitive data and to extend data structures with transactional features instead of encapsulating JavaScript functions. For example, instead of defining `root` directly, a developer can wrap it in a sandbox membrane.

```
1 let sbx5 = new Sandbox(this, Sandbox.DEFAULT);
2 let root = sbx5.wrap(new Node(0, new Leaf(0), new Leaf(0)));
```

Listing 9.40 Wrapping an object in a sandbox.

Now, the global `root` object is wrapped in a sandbox membrane identical to the value visible inside of the sandbox. Proxies guarantee that the new `root` object performs as usual, for example when calling `setValue`.

```
1 setValue(root);
```

Listing 9.41 Applying `setValue` to a sandbox object.

However, as `root` is a sandbox object, it enables a developer to use all sandbox features in addition. After calling `setValue` we can inspect effects, check for changes and conflicts, and revert to its initial status.

10 Sandbox Encapsulation

This chapter presents the design principles underlying DecentJS. Its design is inspired by revocable references [20] and SpiderMonkey's compartment concept [109].

Compartments create different memory heaps for different websites. 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. The motivation is to optimize garbage collection in the SpiderMonkey JavaScript engine.

Our sandbox adapts SpiderMonkey's compartment concept and offers the possibility to run code in isolation to the application state. Proxies implement a membrane and make objects accessible inside of a sandbox. In addition, they provide effect logging and implement features similar to transactional database systems.

10.1 Memory Safety

The implementation of DecentJS builds on two foundations: *memory safety* and *reachability*.

In memory safe programming languages, a reference can be seen as the transferable right to access the public interface of an object. As JavaScript does not support pointer arithmetic, a code fragment can only get access to a certain resource if it gets a reference to that resource. In JavaScript, all resources are accessible via property read and property write operations[1]. Thus, controlling those operations is sufficient to control the resources.

Central to our sandbox is the implementation of a sandbox membrane (proxy membrane) on values that cross the sandbox boundary. The membrane guarantees that each visible object inside of the sandbox is either an object that only appears inside or it is a wrapper for some outside object. Moreover, it supplies effect monitoring and features identity preservation.

10.2 Shadow Objects

Our sandbox redefines the semantics of proxies to implement expanders [110], an idea that allows a client-side extension of properties without modifying the proxy's target.

A sandbox handler consists of two target objects: a proxy target object and a local *shadow object*. The target object acts as a parent object for its proxy whereas the shadow object gathers local modifications. A write operation always takes place on the shadow object. A read operation either forwards the read to the proxy's target object, if the property is not locally modified, or it reads the property from the shadow object. Figure 10.1 illustrates this situation, which is very similar to JavaScript's prototype chain: the sandbox proxy is a child object that inherits everything from its outside (parent) object, whereas modifications only appear inside (locally) on the sandbox proxy.

Figure 10.2 demonstrates the membrane arising from the example in Chapter II. The upper part shows the state that exists after calling `setValue` from Line 1 of Listing 9.6. Doing this creates a proxy for each element of `root`. The proxy object forwards the write operations to the shadow copies whose properties replace their target counterparts. All modifications during the sandbox call are only visible inside of the sandbox.

[1] Each variable access is a property access to JavaScript's scope chain.

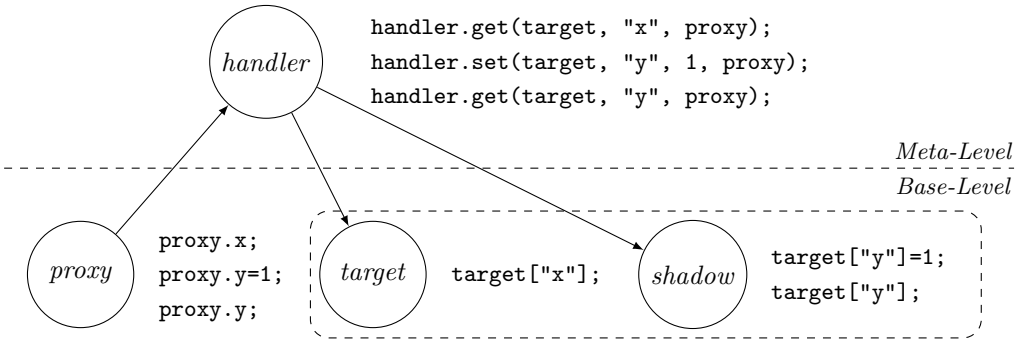


Figure 10.1 Example of a sandbox operation. The property get `proxy.x` invokes the trap `handler.get(target, "x" ,proxy)`, which forwards the operation to the proxy's target. The property set `proxy.y=1` invokes the trap `handler.set(target, "y", 1, proxy)`, which forwards the operation to the proxy's shadow object. The property get `proxy.y` is then also forwarded to the shadow object.

The middle part shows the membrane after extending `root`'s left leaf by a new subtree in line 1 of Figure 9.17. As we already modified `root`'s left leaf inside of the sandbox, the sandbox remembers the previously read data structure and preserves a consistent view to `root`.

The bottom part shows the membrane after calling `appendRight` in line 1 of Figure 9.24. As we only replace `root`'s right leaf, the only wrapped value visible in the sandbox is `root`. This is because proxies only arise on demand. The new node elements are not wrapped in a proxy because they only exist inside of the sandbox.

10.3 Sandbox Scope

Apart from access restrictions, protecting the global state from uncontrolled access through the sandbox membrane is fundamental to guarantee noninterference. To this end, DecentJS relies on `eval`, which is nested in a `with (sbxglobal) { /* body */ }` statement. The `with` statement places the `sbxglobal` on top of the current scope chain while executing `body` and `eval` dynamically rebinds the free variables of its argument to whatever is in scope at its call site. In this setup, which is related to dynamic binding [52], any property defined in `sbxglobal` shadows a variable deeper down in the scope chain.

To guarantee noninterference, we employ a special proxy object in place of `sbxglobal` and override the `hasOwnProperty` trap to always return `true`. When JavaScript traverses the scope chain to resolve non-local variable access, this traversal calls the method `hasOwnProperty` method when reaching the wrapped sandbox global. If the `hasOwnProperty` method always returns `true`, then the traversal stops and the JavaScript engine sends all read and write operations for free variables to the sandbox global. This way, we obtain full interposition, and the handler has complete control over the free variables in `body`.

Figure 10.3 visualizes the nested scopes created during the execution of `setValue` from Line 1 of Listing 9.6. The sandbox global `sbxglobal` is a wrapper for the actual global object, which is used to access `heightOf` and `Math`. The function is nested in an empty closure which provides a fresh scope for local functions and variables. This step is required because JavaScript does not have standalone block scopes such as blocks in C or Java. Variables

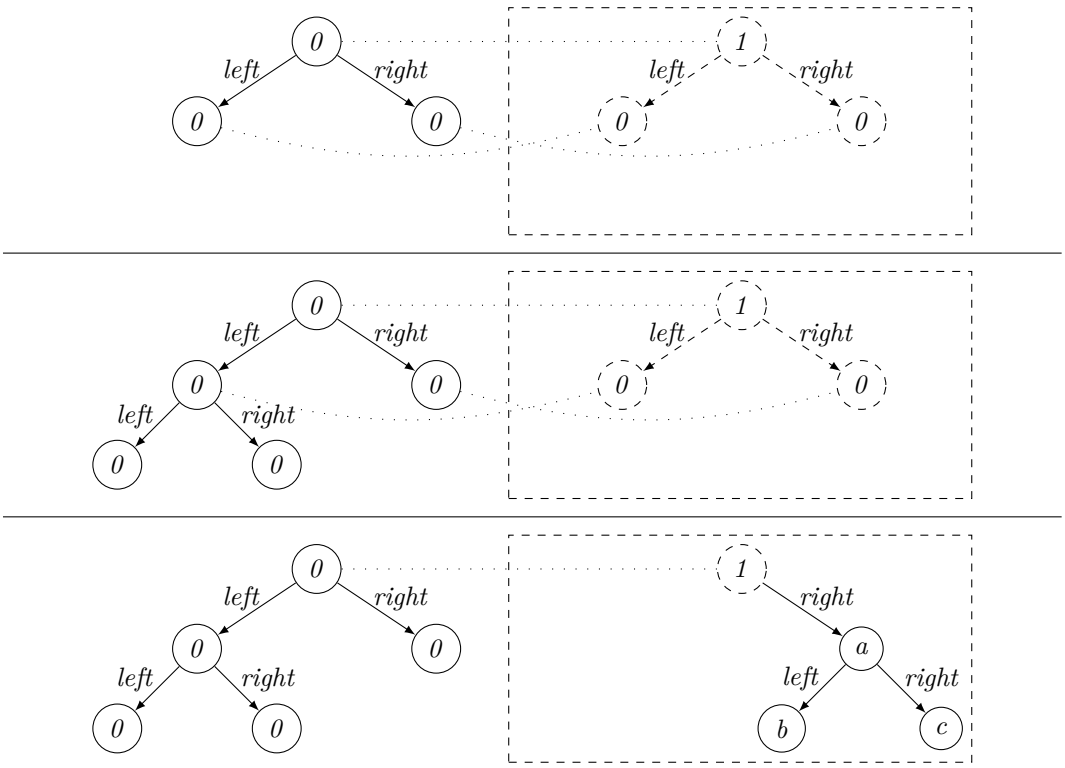


Figure 10.2 Example of shadow objects through a sandbox membrane. The figure shows the binary trees created in Chapter 9. The left side shows the original, whereas the right side shows the visible representation inside of a sandbox. The value set to the `value` field is the node's content. The `left` and `right` fields are represented as edges. Solid lines represent direct references to non-proxy objects, whereas dashed lines represent indirect references and proxies. Dotted lines refer to the corresponding target object. The dashed box symbolized the sandbox.

and named functions[2] created by the sandboxed code end up in this fresh scope. This extra scope guarantees noninterference for dynamically loaded scripts that define global variables and functions. The `"use strict"`[3] declaration in front of the closure puts JavaScript in strict mode, which ensures that the code cannot obtain unprotected references to the global object.

Figure 10.4 shows the situation when instantiating different sandboxes during program execution. Every sandbox installs its own scope with a sandbox global on top of the scope chain. Scripts nested inside are defined with respect to the sandbox global. The sandbox global mediates the access to the outside, for example to JavaScript's global object. A new sandbox always starts with an empty sandbox global. Values can be given to the sandbox by defining a property with that value in the sandbox global.

[2] Function created with `function name() {/* body */}`.
[3] Strict mode requires that each use of `this` inside a function is only valid if either the function was called as a method or a receiver object was specified explicitly using `apply` or `call`.

```
let root = new Node(0, new Leaf(0), new Leaf(0));
let sbx = new Sandbox(this, {/* some parameters */});

with(sbxglobal) {

  (function(){
    "use strict";

    function setValue (node){
      if (node instanceof Leaf) node.value=heightOf(node);
      else {
        node.value=heightOf(node);
        setValue(node.left);
        setValue(node.right);
      }
    }

  })();

}
```

Figure 10.3 Scope chain installed by the sandbox when loading `setValue`. The dark box represents the global scope. The dashed line indicates the sandbox boundary and the inner box shows the program code nested inside.

10.4 Function Recompilation

In JavaScript, a function "remembers the environment in which it was created."[4]. In other words, each function has access to all variables defined in their enclosing scope.

Thus, calling a wrapped function may still cause side effects through their free variables (e.g., by modifying a variable or by calling another side-effecting function). To guarantee noninterference, sandboxing must either erase all external bindings of a function or it has to verify that a function is free of side effects. However, the letter one is not easy for JavaScript as a simple read access might be the call of a side-effecting getter function.

To remove bindings from a function passed through the sandbox membrane DecentJS decompiles the function and redefines it inside of the sandbox. Decompilation relies on the standard implementation of the `Function.prototype.toString` method which returns a string containing the source code of the function. To bypass potential tampering, we use a private copy of `Function.prototype.toString` for this call. After this, we apply `eval` to the resulting string to create a fresh variant of that function with respect to the sandbox environment. As explained in Section 10.3, this application of `eval` is nested in a `with` statement. Decompilation also places the `"use strict"` statement in front.

Functions without a string representation (e.g., native functions like `Object` or `Array`)

[4] The Mozilla documentation, `https://developer.mozilla.org/en/docs/Web/JavaScript/Closures`

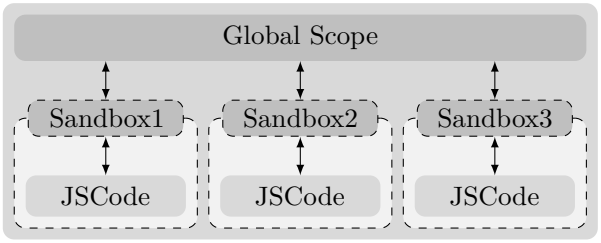


Figure 10.4 Nested sandboxes in an application. The outer box represents the global application state containing JavaScript's global scope. Each sandbox has its own global object and the nested JavaScript code is defined w.r.t. to the sandbox global.

cannot be sanitized before passing them through the membrane. We can either trust these functions or rule them out. To this end, DecentJS may be provided with a white-list of trusted function objects. However, every function remains wrapped in a sandbox proxy to mediate property access.

In addition to normal functions calls, the access to a property that is bound to a `getter` or `setter` function needs to decompile the function before its execution.

10.5 Policies

A *Policy* is a statement that specifies a set of allowed or forbidden operations. For example, a policy can grant read access to a certain resource or it may restrict possible access paths on an object. Most existing sandbox systems come with a facility to define policies.

DecentJS does not provide policies in the manner known from other systems. It only provides the mechanism to implement a fresh (empty) scope and to load values into that environment. However, it provides other techniques to handle fine-grained access control:

- A new sandbox always starts with an empty global object and does not contain references to the outside world. So, the sandboxed code runs in full isolation. Read access to a certain resource can be granted by binding the values when instantiating a new sandbox.
- A sandbox does not cause side effects. Thus, write operations inside are fine. Write access can be granted by committing effects or by using the transparent mode.
- Proxies can easily implement more fine-grained access permission. Each value that crosses the sandbox membrane can be wrapped in another membrane. This membrane can be used to implement access control. For example, one could use *Access Permission Contracts* [64] to restrict the access on objects or *Revocable References* [20] to revoke access to the outside world.

This construct enables that scripts run without restrictions and without recognizing the sandbox.

10.6 DOM Updates

The *Document Object Model* (DOM) is an API for manipulating HTML and XML documents that underlie the rendering of a web page. The DOM provides a representation of the document's content, and it offers methods for changing its structure, style, content, etc. In JavaScript, this API is implemented using special objects, reachable from the `document` object.

The DOM API is not part of the JavaScript standard. Browsers usually extend the runtime environment with a `document` object to grant access to the HTML or XML document. As most JavaScript libraries access the DOM, it is required to make the DOM available when running library code in a sandbox. However, wrapping the `document` interface object in a sandbox membrane leads to a number of limitations:

- By default, DOM nodes are accessed by calling query methods like `getElementById` on the `document` object. Effect logging recognizes these accesses as method calls, rather than as operations on the DOM.
- All query functions are usually special native functions that do not have a string representation. Decompilation is not possible so that using a query function must be permitted explicitly through the whitelist.
- A query function must be called as a method of an actual DOM object implementing the corresponding interface. Thus, DOM objects cannot be wrapped like other objects. They require a special wrapping that calls the method on the correct receiver object. While read operations can be managed in this way, write operations must either be forbidden, or they affect the original DOM.
- With unrestricted write operations, it would be possible to insert new `<script>` elements in a document which loads scripts from the internet and executes them in the normal application state without further sandboxing.

Thus, the `document` object cannot be wrapped for safe use inside of a sandbox. To overcome this limitation, DecentJS provides guest code access to an *emulated* DOM instead of the real thing. We rely on *dom.js*[5], a JavaScript library emulating a full browser DOM, to implement a DOM interface for scripts running in the sandbox. Like the normal DOM, the emulated DOM is merged into the global sandbox object when executing libraries. The sandbox internal DOM can be accessed and modified at will. A special membrane mediates all operations and performs effect logging on all the DOM elements. This construct provides the following features:

- The sandbox provides an interface to the sandbox internal DOM and enables the host program to access all aspects of the DOM. This interface can control the data visible to the library program.
- A host program can load a page template before evaluating library code. This template can be an arbitrary HTML document, like the host's page or a blank web page. As most libraries operate on non-blank page documents (e.g., by reading or writing to a particular element) this template can be used to create an environment.
- Library code runs without restrictions. For example, guest-code can introduce new `<script>` elements to load library code from the internet. These libraries are loaded and executed in the sandbox as well.
- All operations on the interface objects are recorded. Effects can be examined using an API similar to the standard effect API (cf. Section 9.2).
- The host program can perform a fine-grained inspection of the document tree (e.g., it can search for changes and differences). The host recognizes newly created DOM elements, and it can transfer content from the sandbox DOM to the DOM of the host program.

[5] `https://github.com/andreasgal/dom.js/`

10.7 Discussion

Noninterference

DecentJS guarantees integrity and confidentiality. The default "empty" sandbox guarantees to run code in full isolation from the rest of the application, whereas the sandbox global can provide protected references to the sandbox. Proxy objects redirect all write operations to local replications such that sandbox code runs without noticing the sandbox.

Strict Mode

DecentJS runs program code in JavaScript's *strict mode* to prohibit unqualified `this` pointers[6] to the global object. Moreover, JavaScript's strict mode enables to use `eval` and other dynamic features without any restrictions.

Unfortunately, the strict mode semantics is slightly different from the non-strict mode semantics. Assuming strict mode may lead to a dysfunctional code in the sandbox if the code fragment requires non-strict JavaScript. However, this assumption is less restrictive than prohibiting JavaScript's dynamic features, as it is done by other techniques.

Function Decompilation

Decompiling a function closure reopens the closure and removes all external binding of that function. Decompilation may change the meaning of a function as it rebinds the function to whatever is in the scope when defining the function. To preserve the semantics it requires that all free variables are imported into the sandbox. The new closure formed within the sandbox may be closed over variables defined in that sandbox.

Unfortunately, this is a manual task. Only "pure functions"[7] can be decompiled without further attention. As every property read may be the call of a side-effecting getter function, decompilation is unavoidable to guarantee noninterference.

Native Functions

Decompilation requires a string that contains the source code of that function. Unfortunately, calling the standard `toString` method from `Function.prototype` does not work for all functions.

- A native function does not have a string representation. Thus, native functions must either be trusted or forbidden. White-listing of functions can be adjusted.
- The `Function.prototype.bind` method of a JavaScript function creates a new bound function with the same body. Even though both functions contain the same body, the new function loses its string representation. Decompilation of that function is not possible. However, as bound values are not determinable, this behavior is expected.

Object, Array, and Function Initializer

In JavaScript, some objects can be constructed using a *literal notation* (initializer notation). Prominent examples are objects (using `{}`), array objects (using `[]`), and function objects

[6] An unqualified `this` pointer arises from calling a function closures with an undefined `this` value. In this case, `this` points to the global object.

[7] A pure function is a function that only maps its input into an output without causing any observable side effects.

(using the named or unnamed function expression, e.g. `function () {}`). Using the literal notation circumvents all restrictions the sandbox imposes on the `Object`, `Array`, and `Function` constructors. The new objects are directly created from that constructors.

Unfortunately, it is not possible to intercept the construction of objects using the literal notation. All created objects inherit directly from the prototype objects `Object.prototype`, `Array.prototype`, and `Function.prototype`. Thus, using the literal notation enables unprotected read access to the corresponding prototype objects.

However, we will never get direct access to the prototype object itself (using `Object.getPrototypeOf` can be prevented) and we are not able to modify the prototype object. Writes to a JavaScript object always affect the object itself and are never forwarded to the prototype object.

Even though the prototype objects only contain uncritical functions, a global (not sandboxed) script could add sensitive data or a side-effecting function to one of the prototype objects. Thus it can bypass access to unprotected data or a side-effecting function. DecentJS only guarantees full-noninterference if all scripts are packed in a sandbox and only the sandbox itself is allowed to be placed in the host application.

Function Constructor

The built-in function constructor `Function` creates a new function object based on the given arguments. In contrast to function statements and function expressions, the function constructor ignores the surrounding scope and always returns a function that is created in the global scope.

To prevent this, DecentJS never grants unwrapped access to JavaScript's global `Function` constructor even if the constructor is whitelisted as a safe native function. A special wrapping surrounding the `Function` constructor intercepts the construction process and creates a new function with respect to the sandbox global.

11 Experimental Evaluation

This chapter reports on our experience with applying DecentJS to benchmark programs from the Google Octane 2.0 Benchmark Suite[1]. Octane measures a JavaScript engine's performance by running a selection of complex and demanding programs. Each program focuses on a special purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing, and compilation, etc.

11.1 The Testing Procedure

For benchmarking, we wrote a new start script that loads and executes each benchmark program in a fresh sandbox. By setting the sandbox global to the standard global object, we ensure that each benchmark program can refer to properties of the global object as needed. As sandboxing wraps the global object in a membrane, it mediates the interaction of the benchmark program with the global application state.

All benchmarks were run on a machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory. All example runs and measurements reported in this paper were obtained with the SpiderMonkey JavaScript engine. All runtime measurements were taken from a deterministic run, which requires a predefined number of iterations[2], and by using a warm-up run.

11.2 Results

Table 11.1 contains the runtime values for all benchmark programs in two different configurations and Table 11.2 lists the readouts of some internal counters.

As expected, the runtime increases when running a benchmark in a sandbox. While some programs like *EarleyBoyer*, *NavierStrokes*, *pdf.js*, *Mandreel*, and *Box2DWeb* are heavily affected, others are only slightly affected: *Richards*, *Crypto*, *RegExp*, and *Code loading*, for instance. Unfortunately, *DeltaBlue* and *zlib* do not run in our sandbox: *DeltaBlue* attempts to add a new property to the global `Object.prototype`. This modification is only visible inside of the current sandbox and not to objects created using literal notation. The *zlib* benchmark uses an indirect call[3] to `eval` to write objects to the global scope, which is not allowed by the ECMAScript 6 (ECMA-262) specification. Another benchmark, *Code loading*, also uses an indirect call to `eval`. A small modification makes the program compatible with the normal `eval`, which can safely be used in our sandbox.

In the first experiment, we turn off effect logging, whereas in the second one it remains enabled. Doing so separates the performance impact of the sandbox system (proxies and shadow objects) from the impact caused by the effect system. From the running times, we find that the sandbox itself causes an average slowdown by a factor of 8.01 (overall benchmarks).

For better understanding, Table 11.2 lists some numbers of internal counters. The numbers indicate that the heavily affected benchmarks (*RayTrace*, *pdf.js*, *Mandreel*, or

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

[2] Programs run either for one second or a predefined number of iterations. If there are too few iterations in one second, it runs for another second.

[3] An indirect call invokes the `eval` function by using a name other than `eval`.

Benchmark	Baseline	Sandbox w/o Effects		Sandbox w Effects	
	time (sec)	*time (sec)*	*slowdown*	*time (sec)*	*slowdown*
Richards	9	12	1.33	15	1.67
DeltaBlue	9	-	-	-	-
Crypto	18	42	2.33	88	4.89
RayTrace	9	74	8.22	498	55.33
EarleyBoyer	19	202	10.63	249	13.11
RegExp	6	9	1.5	12	2
Splay	3	19	6.33	33	11
SplayLatency	3	19	6.33	33	11
NavierStokes	3	56	18.67	61	20.33
pdf.js	7	113	16.14	778	111.14
Mandreel	8	151	18.88	483	60.38
MandreelLatency	8	151	18.88	483	60.38
Gameboy Emulator	4	17	4.25	26	6.50
Code loading	8	11	1.38	12	1.50
Box2DWeb	4	145	36.25	1,302	325.50
zlib	7	-	-	-	-
TypeScript	26	61	2.35	328	12.62
Total	135	1.082	8.01	4,401	32.60

Table 11.1 Timings from running the Google Octane 2.0 Benchmark Suite. The first column **Baseline** gives the baseline execution times without sandboxing. The column **Sandbox w/o Effects** shows the time required to complete a sandbox run without effect logging and the relative slowdown (Sandbox time/Baseline time). The column **Sandbox w Effects** shows the time and slowdown (w.r.t. Baseline) of a run with fine-grained effect logging.

Box2DWeb) perform a huge number of effects. In absolute times, raw sandboxing causes a runtime deterioration of 3μs per sandbox operation (effects) (11μs with effect logging enabled). For example, the *Box2DWeb* benchmark requires 145 seconds to complete and performs 132,722,198 effects on its membrane. Its baseline needs 4 seconds. Thus, sandboxing takes an additional 141 seconds. Hence, there is an overhead of 1μs per operation (10μs with effect logging enabled).

To allow a better comparison of the benchmark results, Table 11.3 shows the score values reported by the benchmark suite. Octane reports its result in terms of a score that is inversely proportional to the runtime of a program. Moreover, the score value of the *SplayLatency* and *MandrealLatency* benchmark measures the interrupts of the garbage collection subsystem. As the benchmarks penalize long interrupt with a lower score, the score value indicates a more frequent and a more uneven use of the garbage collection.

11.3 Memory Consumption

Table 11.4, Table 11.5, and Table 11.6 show the memory consumption recorded when running the Google Octane 2.0 Benchmark Suite. Table 11.4 shows the memory usage when running the benchmark programs without sandboxing, whereas Figure 11.5 and Table 11.6 shows the memory usage of a run with raw sandboxing and of a run with effect logging.

The numbers indicate that there is no significant increase in the memory consumed. For example, the difference of the virtual memory size ranges from -126MByte to 40MByte for a raw sandbox run and from -311MByte to +158MByte for a full run with fine-grained effect logging. Obviously, effect logging slightly increases the memory consumption as it requires

Benchmark	Objects	Effects	Size of Effect List		
			Reads	*Writes*	*Calls*
Richards	14	492073	20	2	5
DeltaBlue	-	-	-	-	-
Crypto	21	4964248	29	2	11
RayTrace	18	51043282	26	3	8
EarleyBoyer	33	4740377	42	8	6
RegExp	16	296995	23	2	6
Splay	16	1635732	23	2	8
SplayLatency	16	1635732	23	2	8
NavierStokes	15	4089	21	2	6
pdf.js	36	77665629	59	8	21
Mandreel	31	39948598	50	2	21
MandreelLatency	31	39948598	50	2	21
Gameboy Emulator	28	1225935	42	2	16
Code loading	12417	107481	50	2	13
Box2DWeb	28	132722198	38	2	14
zlib	-	-	-	-	-
TypeScript	23	27518481	34	2	9
Total	12743	383949448	530	43	173

Table 11.2 Numbers from internal counters. Column **Objects** shows the numbers of wrapped objects and column **Effects** gives the total numbers of effects. Column **Size of Effect List** lists the numbers of *different* effects after running the benchmark. Column **Reads** shows the number of read effects distinguished from the number of write effects (Column **Writes**) and distinguished from the number of call effects (Column **Calls**). Multiple effects to the same field of an object are counted as one effect.

to store all the effects. For the effect-heaviest benchmark *Box2D* we find that the *virtual memory size* rises from 157MByte (raw run) to 197MB (full run with effect logging).

To sum up, the results from the tests indicate that there is no significant increase in memory consumption, but the garbage collector runs more frequently. Unfortunately, more frequent garbage collector calls impact the total execution time of a benchmark program.

11.4 Notes

We use Octane as it aims to measure a JavaScript engine's performance by running a selection of complex and demanding programs found in large, real-world web applications. We claim that it is the heaviest kind of benchmark for such a sandbox system.

The numbers that we obtain give an insight into the performance of our sandbox implementation. However, the current setting needs to place the whole benchmark in a closure. Because there are currently no large programs that allow fine-grained sandboxing the numbers cannot be used to predict the performance impact of a more fine-grained use by human programmers.

Benchmark	Sandbox w/o Effects	Sandbox w Effects	Baseline
Richards	4825	4135	6552
DeltaBlue	-	-	6982
Crypto	2418	1131	5669
RayTrace	1387	179	9692
EarleyBoyer	954	767	10345
RegExp	911	1139	1535
Splay	1268	713	8676
SplayLatency	3630	1818	12788
NavierStokes	989	890	15713
pdf.js	434	63	8182
Mandreel	346	106	9102
MandreelLatency	2518	526	12526
Gameboy Emulator	6572	4780	31865
Code loading	7348	6000	9136
Box2DWeb	453	50.1	18799
zlib	-	-	42543
TypeScript	4554	792	12588

Table 11.3 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Block **Sandbox w/o Effects** contains the score values of a raw sandbox run without effect logging, whereas block **Sandbox w Effects** contains the score values of a full run with fine-grained effect logging. The last column **Baseline** gives the baseline scores without sandboxing.

Benchmark	Baseline			
	Virtual *size*	**Resident** *size*	**Text/Data** *size*	**Shared** *size*
Richards	134	19	109	5
DeltaBlue	-	-	-	-
Crypto	225	105	201	6
RayTrace	148	31	124	5
EarleyBoyer	500	363	476	6
RegExp	226	108	202	6
Splay	535	416	511	6
SplayLatency	535	416	511	6
NavierStokes	141	24	116	5
pdf.js	316	169	292	6
Mandreel	305	182	280	6
MandreelLatency	305	182	280	6
Gameboy Emulator	194	62	170	6
Code loading	268	142	243	6
Box2DWeb	157	53	132	5
zlib	-	-	-	-
TypeScript	473	369	448	6

Table 11.4 Memory usage when running the Google Octane 2.0 Benchmark Suite without sandboxing. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. All values are in MByte.

Benchmark	**Sandbox w/o Effects**							
	Virtual		**Resident**		**Text/Data**		**Shared**	
	size	*diff.*	*size*	*diff.*	*size*	*diff.*	*size*	*diff.*
Richards	135	+1	20	+1	110	+1	5	0
DeltaBlue	-	-	-	-	-	-	-	-
Crypto	232	+7	106	+1	208	+7	6	0
RayTrace	148	+0	31	+0	124	+0	6	+1
EarleyBoyer	374	-126	272	-91	350	-126	6	0
RegExp	224	-2	107	-1	200	-2	6	0
Splay	466	-69	352	-64	422	-89	6	0
SplayLatency	466	-69	352	-64	422	-89	6	0
NavierStokes	134	-7	18	-6	109	-7	5	0
pdf.js	274	-42	123	-46	250	-42	6	0
Mandreel	263	-42	133	-49	239	-41	5	-1
MandreelLatency	263	-42	133	-49	239	-41	5	-1
Gameboy Emulator	188	-6	60	-2	163	-7	6	0
Code loading	259	-9	138	-4	234	-9	6	0
Box2DWeb	197	+40	97	+44	172	+40	6	+1
zlib	-	-	-	-	-	-	-	-
TypeScript	424	-49	325	-44	428	-20	6	0

Table 11.5 Memory usage of a raw sandbox run without effect logging. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. Sub-column *size* shows the size in MByte and sub-column *diff.* shows the difference to the baseline (Sandbox size - Baseline size) in MByte.

Benchmark	**Sandbox w Effects**							
	Virtual		**Resident**		**Text/Data**		**Shared**	
	size	*diff.*	*size*	*diff.*	*size*	*diff.*	*size*	*diff.*
Richards	167	+33	54	+35	142	+33	6	+1
DeltaBlue	-	-	-	-	-	-	-	-
Crypto	209	-16	93	-12	184	-17	6	0
RayTrace	181	+33	63	+32	157	+33	6	+1
EarleyBoyer	189	-311	86	-277	195	-281	6	0
RegExp	224	-2	104	-4	200	-2	6	0
Splay	424	-111	321	-95	399	-112	6	0
SplayLatency	424	-111	321	-95	399	-112	6	0
NavierStokes	134	-7	19	-5	109	-7	5	0
pdf.js	272	-44	116	-53	248	-44	6	0
Mandreel	347	+42	160	-22	323	+43	6	0
MandreelLatency	347	+42	160	-22	323	+43	6	0
Gameboy Emulator	214	+20	84	+22	190	+20	6	0
Code loading	262	-6	139	-3	238	-5	6	0
Box2DWeb	191	+34	72	+19	166	+34	6	+1
zlib	-	-	-	-	-	-	-	-
TypeScript	631	+158	493	+124	607	+159	6	0

Table 11.6 Memory usage of a full run with fine-grained effect logging. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. Sub-column *size* shows the size in MByte and sub-column *diff.* shows the difference to the baseline (Sandbox size - Baseline size) in MByte.

Transparent Object-Proxies for JavaScript

12 Proxies, Membranes, and Contracts

As the implementation of proxies and membranes is already discussed in Section 3, we will just briefly recall the use of proxies to implement contract systems. Proxies implement contracts in Racket's contract framework [45, Chapter 7], in Disney's JavaScript contract system *Contracts.js* [30], in JSConTest2 for JavaScript [64], and in the TreatJS [68] contract framework for JavaScript.

When implementing contracts, proxies are important to state properties that cannot be checked immediately: for example invariants on objects or pre- and postconditions on functions and methods. To demonstrate contract checking, consider the following function `plus` which accepts two arguments of type number and promises to return a value of type number.

```
let plus = Contract.assert(function plus(x, y) {
  return (x+y);
}, Contract.Function([typeNumber, typeNumber], typeNumber));
```

Listing 12.1 Example of a delayed contract.

We call a function contract *delayed*, because asserting it *to a function* does not immediately signal a contract violation. Asserting a function contract amounts to asserting the domain contract to the arguments of each call of the function and asserting the range contract to the return of each call.

The assertion of a delayed contract may be implemented by wrapping the function in a proxy. The proxy handler contains the contract and implements a trap to mediate the use of the function and to assert its contract when the function is used. The following example sketches the implementation of a delayed contract assertion that checks for number values.

```
let handler = {
  apply: function (subject, thisArg, argumentsArg) {

    // check if arguments are of type number
    if(!(typeof argumentsArg[0] === 'number')) throw new TypeError();
    if(!(typeof argumentsArg[1] === 'number')) throw new TypeError();

    // call the target function
    const result = Reflect.apply(subject, thisArg, argumentsArg);

    // check if result is of type number
    if(!(typeof result === 'number')) throw new TypeError();

    // return the result
    return result;
  }
};
let plus = new Proxy(function plus(x, y) {
  return (x+y);
}, handler);
```

Listing 12.2 Sketch Implementation of a Delayed Contract.

The proxy installs a handler that intercepts all function applications. It's `apply` trap gets invoked when the proxy is called as a function. The arguments passed to the apply method

are the target object (function `plus` in this example), the `this` argument for the call, and an array-like object containing the arguments for the call.

The trap method first checks the argument values before it proceeds with the usual operation. Later it checks the value that returns from forwarding the call to the target object. In both cases, it throws a type error if either an argument or the return is not a number value. The addition of this contract does not change the normal program execution of a well-behaved program.

Both contract systems, Racket and TreatJS, implement *delayed* contracts on objects and functions with specific wrapper objects, Racket's chaperones and impersonators [100] and JavaScript proxies [20], respectively. Dynamic contract monitoring then uses the proxy object in place of the target object. However, as it cannot delete or rewrite existing references to the original function, target and proxy object may occur in the same execution environment. But, this may change the semantics of a program, and thus it violates a ground rule for monitoring: a monitor should never interfere with the execution of a program conforming to the monitored property.

12.1 Opaque Proxies

A proxy is an object that mediates access to an arbitrary target object. The objective of introducing a proxy is to extend or restrict the functionality of the underlying object. Proxies are widely used to perform resource management, to access remote objects, to impose access control [20, 64], to implement contract checking [100, 41, 30, 68], to restrict the functionality of an object [100], to enhance the interface of an object [110], to implement dynamic effects systems [64], for meta-level extension, for behavioral reflection, for security [2], and for concurrency control [80, 5, 12]. Chapter 3 gives a detailed introduction to JavaScript Proxies.

Ideally, a program should not be able to distinguish a proxy from a non-proxy object, i.e., running a program with an interposed proxy should lead to the same outcome as running the program with the target object unless the proxy imposes restrictions. For that reason, the JavaScript proxy API [20, 33] does not provide a function that checks whether an object is a proxy or not, it does not offer traps for all possible operations on proxies, and it places several restrictions on traps to avoid breaking object invariants [20].

Unfortunately, the JavaScript Proxy API treats a proxy of a target object as a new object different from its target. Each proxy has its own identity, different from all other proxy or non-proxy objects. The definition of object equality [33, Section 7.2.13] says: *If x and y are the same object value, return true.*

This means that proxies are *opaque* with respect to object equality. However, given opaque proxies an equality test can be used to distinguish a proxy from its target, as demonstrated in the following example:

```
1 let target = { /* some object */ };
2 let handler = { /* empty handler */ };
3 let proxy = new Proxy (target, handler);
4 proxy === target; // evaluates to false
```

Listing 12.3 Distinguish opaque proxies.

Even though `target` and `proxy` behave identical, the equality test fails. The double and triple equal operators (`==`, `===`) only evaluate to *true* if and only if both operands refer to the same object. Thus, in a program that uses object equality, the introduction of a proxy can lead to a program failure (without even invoking an operation on the proxy) when a proxy is directly used in place of the target object.

12.2 A Discussion of different Use-Cases

JavaScript proxies are used for Disney's JavaScript contract system *Contracts.js* [30], to implement higher-order contracts in TreatJS [68], to enforce Access Permission Contracts [64], for security [2], as well as for revocable references and membranes [20].

However, JavaScript proxies introduce a subtle problem. Because a target object may have multiple proxy objects, which are all different from the target, a single target object may obtain multiple identities. Object equality for opaque proxies works well under the assumption that proxies and their target objects are never part of the same execution environment. However, it turns out that this assumption is not always appropriate. One prominent use case is the implementation of a contract system.

Unfortunately, the current proxy implementation is committed to particular use cases which makes it hard to adapt it to uses with different requirements. We now discuss three such use cases (object extension, access-restricting membranes, and contract checking) in the context of the JavaScript Proxy API [20, 33], identify their shortcomings, and propose a solution.

12.2.1 Use Case: Object Extension

A common use case of proxies is to extend or redefine the semantics of the underlying target object. For example, a handler may throw an error instead of returning `undefined` for a non-existing property, it may implement an *expander* [110] to allow client-side extension of properties without modifying its origin, or it may redirect different operations to different targets (for example to implement placeholders or shadow objects).

In this cases, the proxy behaves different to the targets and using the proxy may lead to a completely different outcome. As proxy and target object implement different semantics, they should not be confused.

12.2.2 Use Case: Access Control

JavaScript proxies implement access control wrappers like *revocable references* [20, 81], the motivating use case for membranes (cf. Section 3.2). The idea of a revocable reference is to only ever pass a proxy to an untrusted piece of code, e.g., a mashup. Once the host application deems that the mashup has finished its job, it revokes the reference which detaches the proxy from its target. Identity preserving membranes extend this method recursively to all objects reachable from a target object.

Opaque proxies are suitable for implementing revocable references and identity preserving membranes, but not strictly required. The JavaScript proxy API is tailored to uses where access is strictly compartmentalized. The host application only sees the original objects whereas the mashup only sees proxies. Furthermore, the implementation of revocable references and identity preserving membranes ensures that there is at most one proxy for each original object. For this reason, each compartment has a consistent view where object references are unique.

Unlike opaque proxies, membranes based on transparent proxies are always identity preserving and do not need a map to ensure this. However, a map that reveals transparent proxies would improve the runtime and space efficiency as it prevents the system from re-wrapping the same object again and again.

12.2.3 Use Case: Contracts

Proxies already implement contracts in JavaScript [30, 68]. Contracts impose restrictions a programmer regards as the correct execution of a program. For example, a contract may require a function to be called with a particular type or an object property to contain positive numbers.

During maintenance, the programmer may add contracts to sensitive data, for example to the arguments of a function. In this scenario, the program execution ends up in a mix of objects with and without contracts. Furthermore, the same object may appear with and without a contract in the same execution environment, and different proxies may implement different contracts for the same target. As the original object may be compared with its contracted counterpart (e.g., by using `===`) some equality comparisons would flip their outcome and thus change the semantics of a program.

Consequently, the Racket implementation provides *transparent proxies* [100], which are indistinguishable (in respect to their structural equality) from their target object, recursively.

The following example shows the desired behavior of hypothetical transparent proxies in JavaScript.

```
var proxyA = new Proxy(target, handler);
var proxyB = new Proxy(target, handler);
var proxyC = new Proxy(proxyB, handler);
(target==proxyA); // returns false, should be true
(target==proxyC); // returns false, should be true
```

Listing 12.4 Desired behavior of a transparent proxy.

Transparent proxies are not distinguishable from their base target. Comparing a transparent proxy with an object should lead to the same outcome as comparing the object with the proxies base target.

```
(proxyA==proxyB); // returns false, should be true
(proxyA==proxyC); // returns false, should be true
```

Listing 12.5 Desired behavior of a transparent proxy (cont'd).

Also, comparing two transparent proxies of the same target should lead to the same outcome as comparing the proxies base targets.

12.2.4 Assessment

Neither the opaque nor the transparent proxy implementation can be labeled as right or wrong without further qualification. Each implementation is appropriate for a particular use case and leads to undesirable behavior in another use case.

Transparent proxies can safely be used to implement a revocable membrane. The implementation can use a weak map that consequently returns the proxy if a key value fits. There is no need to wrap a proxy again. In contrast, a contract system requires transparent proxies by default. Otherwise, it would influence the normal program execution. But contract systems might also need to distinguish different contracted versions of the same target object. Completely indistinguishable proxies avoid this.

It is also clear that the behavior of equality is not something that should be left to the whim of the programmer. For example, equality on objects should be an equivalence relation, which means that the equality operations `==` and `===` must not be trapped (see also [21]). Thus, the current state of affairs in JavaScript is entirely justified, but it is not suitable to implement contract systems. If a program uses object equality, then adding contracts based

on opaque proxies may change the behavior of well-behaved programs. Hence, we explore some alternative designs that would suit all use cases.

12.3 An Analysis of Object Comparisons

This section considers whether the contract implementation based on opaque proxies affects the meaning of *realistic* programs. To this end, we consider a typical maintenance scenario where a programmer adds a contract to a particular piece of code. Its objective is to count and classify proxy-object comparisons with regard to their influence on the program execution.

For example, consider the following function `isTarget` which checks it's argument to be `target`.

```
let target = { /* some object */ };
function isTarget(arg) {
  return arg===target;
}
```

Listing 12.6 Definition of function `isTarget`.

Furthermore, let `Q` be some delayed contract and consider function `cmp` which takes two parameters, `f` and `x`, and which returns the result of applying `f` to `x`.

```
let cmp = Contract.assert(function cmp(f, x) {
  return f(x);
}, Contract.Function([Contract.Function([Q],typeBoolean), Q], typeBoolean);
```

Listing 12.7 Definition of function `cmp`.

Now, let's use `cmp` as follows:

```
cmp(isTarget, target);
```

Listing 12.8 Using `cmp` to compare `target`.

Calling `cmp` wraps `isTarget` and `target` in a proxy that enforces the given delayed contracts `Contract.Function([Q],typeBoolean)` and `Q`. Unfortunately, wrapping `isTarget` does not affect the bound variables in the function's environment. However, inside of `cmp`, `obj` is wrapped in a proxy and calling `f(x)` may wrap it one more time in `Q`. Thus, comparing `arg==target` yields `false` instead of `true`, the result before installing the contract on `cmp`.

Consequently, if a program uses object equality, then adding contracts based on opaque proxies may change the behavior of well-behaved programs.

12.3.1 The Experiment

To obtain the number of object comparisons that occur during a program execution we instrument the JavaScript engine to count and classify proxy-object comparisons.

Our model is a recursive object wrapper that simulates a simple contract system by wrapping the arguments of a function. The proxy handler solely forwards the operation to the target object. To adapt existing implementations, we use an identity-preserving membrane that maintains aliasing and avoids chains of nested proxies.

The subject programs are again taken from the Google Octane 2.0 Benchmark Suite [85]. A preceding source-to-source compiler generates a new file for each function expression in a benchmark program, each of which wraps exactly one function in our contract system. Separated programs enable us to reason about different proxies from different contracts.

After compiling, the modified benchmark programs were executed in a special engine that treats proxies as transparent. While not influencing the normal program execution, our engine counts and classifies all object-proxy comparisons. All numbers were taken from a *deterministic run*, a predefined setting which requires a fixed number of iterations, and by using a warm-up run.

12.3.2 A Classification of Proxy-Object Comparisons

Before discussing the results, we introduce a classification of object comparisons (equality tests). We only consider comparisons between *two* objects, either of them needs to be a proxy. All other comparisons are not influenced by introducing proxies.

Let t be an arbitrary target object and $M(t)$ a target wrapped in a membrane M.

Type-I: $M(t_1) == t_2$ or $M_1(t) == M_2(t_2)$. All equality tests between a proxy object and another object, which might be either a native (non-proxy) object or a proxy-object from another membrane. Tests in this class *always* return `false` when using opaque proxies, whereas the result of a transparent proxy depends on the proxy's target.

Type-Ia: $t_1 \neq t_2$. Subclass of **Type-I**, where the proxy's target object t_1 is different from the other object or the other proxy's target object, respectively. Opaque and transparent proxies yield the same result, `false`, but for different reasons.

Type-Ib: $t_1 = t_2$. Subclass of **Type-I**, where the proxy's target object t_1 is identical to the other object or the other proxy's target object. An equality test of this type yields `false` when using opaque proxies, whereas transparent proxies yield `true`.

Type-II: $M(t_1) == M(t_2)$ All equality tests between two proxy objects from the same membrane. Tests in this class *always* return `false` when using opaque proxies without an identity-preserving membrane. With an identity-preserving membrane, or with transparent proxies, the result depends on the proxy's target.

Type-IIa: $t_1 \neq t_2$. Subclass of **Type-II**, where the proxies' target objects differ. Opaque and transparent proxies yield the same result, `false`, but for different reasons.

Type-IIb: $t_1 = t_2$. Subclass of **Type-II**, where the proxies' target objects are the same. An equality test in this class yields `false` when using opaque proxies without an identity-preserving membrane, whereas transparent proxies yield `true`. Opaque proxies from an identity-preserving membrane also yield `true`.

Identity preserving membranes use weak maps to manage already wrapped objects. This serves two purposes: First, it avoids re-wrapping of a proxy in the same membrane and, second, it can be used to reflect object identity. This gives better performance because the VM does not need to evaluate an entire chain of proxies to reach the base target object and we do not have multiple proxies for the same target.

In this setting, we count equality tests between two proxy objects from different membranes in category **Type-I**, because different contracts implement different membranes and the mechanism that preserves the identity does not work when using different membranes.

12.3.3 Numbers of Comparisons involving Proxies

The table in Figure 12.1 summarizes the numbers of equality tests involving proxy objects. Equality tests between two native objects and all tests involving primitive values (e.g., boolean values, numbers, null, or undefined) are omitted from the result because they are not influenced by introducing proxy objects. Benchmark programs not listed in this table

Benchmark	Total	Type-I		Type-II	
		Type-Ia	Type-Ib	Type-IIa	Type-IIb
DeltaBlue	144126	29228	1411	33789	79698
RayTrace	1075606	0	0	722703	352903
EarleyBoyer	87211	8651	6303	53389	18868
TypeScript	801436	599894	151297	20500	29745

Table 12.1 Number of equality tests involving object proxies. Column **Total** contains the total number of comparisons. Column **Type-I** lists the comparisons of **Type-I**, divided into the two subclasses **Type-Ia** and **Type-Ib**. Column **Type-II** shows the number of **Type-II** comparisons, divided into the subclasses **Type-IIa** and **Type-IIb**.

do not contain equality tests with a proxy object. However, they still contain comparisons between native objects or between objects and primitive values[1].

The numbers cover all different types of comparisons: *equal* (`==`), *not equal* (`!=`), *strict equal* (`===`), and *strict not equal* (`!==`). Therefore, the term "the result is *true*" is used in a generalized sense and means that *equal* and *strict equal* yields *true*, whereas *not equal* and *strict not equal* yields *false*. Not covered by the numbers are internal object comparisons of build-in native objects, for example, weak maps and weak sets, that use object equality when adding or deleting an entry. However, none of the benchmarks uses one of them. The numbers show that a total number of 159011 equality tests (sum of **Type-Ib**) flip from `true` to `false` when replacing objects with their opaque proxies.

The numbers also show that an identity-preserving membrane ensures that 481214 equality tests (sum of **Type-IIb**) evaluate correctly. However, if it is not possible to use a membrane that preserves the identity, then further 830381 equality tests would flip from `true` to `false`.

Furthermore, the obtained numbers show that 1468154 equality tests (sum of all **Type-Ia** and **Type-IIa** numbers) are not influenced by introducing opaque proxies. However, the reason for this result is different. An opaque proxy compared with a native object (or with another opaque proxy object) yields `false` because the proxy is not equal to the other object, whereas a transparent proxy generates `false` because the proxy's target object is not equal to the other object (to the other proxy's target object).

12.4 Summary

The evaluation shows that a significant number of object comparisons fails when mixing opaque proxies and target objects in the same execution environment, for example when implementing a contract system with opaque proxies. A total number of 159011 (640225 without identity preserving membranes) out of 2108369 equality tests flip their outcome and may, therefore, influence the normal program execution. Identity preserving membranes will minimize this number, but they are not able to prevent programs from incorrect evaluations.

However, the numbers also show that the majority of object comparisons is not affected by opaque proxies. Reason for the small number of flipped equality tests is the careful handling of object comparisons in JavaScript. Results from other experiments show that approximately 6% of all equality tests involve two objects. The other comparisons either check an object against null, test for undefined values or compare primitive values.

[1] JavaScript developers frequently use the double and triple equality operator to test for `null` and `undefined`.

13 Design Alternatives for Proxy Equality

JavaScript provides two kinds of comparison operators: Strict equality operators (e.g. `===` and `!==`) and type-converting equality operators (e.g. `==` and `==`). Furthermore, there are relational comparisons (e.g., `<=`) and some build in operations that rely on object equality, for example, the `get` and `set` operations of a map that computes a hash value for an object when using the object as a key value.

Unfortunately, JavaScript proxies are not mentioned in the definition of object equality: *If x and y are the same Object value, return true.* [33, 8., Section 7.2.13]. Thus, proxies are opaque by choice of their implementation, and each proxy has its own identity, different from all other proxy or non-proxy objects. However, in some use cases, this can lead to a program failure when mixing proxies and their target objects in the same execution environment.

This chapter discusses different design alternatives to overcome this limitation.

13.1 Invariants for Equality

Before discussing design alternatives for proxy equality, let's consider two different definitions of equality. Equality is when things are the same (in some particular way). Formally, there are two ways to define object equality:

Equivalence Relation Viewed as a relation, object equality is both, the finest equivalence relation and a partial order on objects. An equivalence relation $\equiv$ is a binary relation that is for all objects a, b, and c:

- reflexive: $a \equiv a$,
- transitive: $a \equiv b \wedge b \equiv c \Rightarrow a \equiv c$
- symmetric: $a \equiv b \Leftrightarrow b \equiv a$

A partial order $\equiv$ is a binary relation that is for all objects a, b, and c:

- reflexive: $a \equiv a$,
- transitive: $a \equiv b \wedge b \equiv c \Rightarrow a \equiv c$
- antisymmetric: $a \equiv b \wedge b \equiv a \Rightarrow a = b$

Now, equality can be defined using $\equiv$: a is equals to b if and only if $a \equiv b$.

Observational Equivalence Observational equivalence is a property that states that the underlying entities cannot be distinguished by observation. For example, the sets $\{a, b\}$ and $\{b, a\}$ are observationally equivalent when using the membership operation $\in$ as observer. Both sets are indistinguishable and produce the same results.

Programming languages like Java, Objective C, C#, Python, and Racket come with different operations for testing equality. *Referential equality* compares memory addresses (pointers) and is, therefore, an equivalence relation. One object is equal to another object if they point to the same address, and unequal if they do not. *Structural equality*, on the other hand, compares the content of an object and tests for observational differences. Two objects are structurally equal if and only if they are indistinguishable on their observable components.

Two examples of this are Racket's `equal?` and `eq?` functions, where `equal?` implements structural equality and `eq?` test for pointer equality, or Java's `==` operator and the `equals?` method in `java.lang.Object`.

In some programming languages (e.g., Java) it is up to the user to define the basis for structural equality. Two objects a and b are observable equivalent if one can substitute a for

b in a certain context without observable differences, i.e., a and b produces the same results. Thus, sometimes two objects are structurally equal even if they have different types[1].

Both definitions are independent of one another. Structural equality did not require that the operands are pointer equal, but in most programming languages it holds that pointer equality implies structural equality, but this is not absolutely necessarily required.

13.2 Object Equality in JavaScript

For objects, JavaScript provides only equality operators that test for referential equality. To test for structural equality, JavaScript developers either use the built-in `Object.prototype.toString` or `JSON.stringify` method to produce and compare a string of that object or they have to write their own function to compare such objects. However, using `toString` is disadvantageous, because the returned string depends on the order of added properties.

Unfortunately, none of the equality operators in JavaScript forms an equivalence relation. The strict equality operator is not reflexive, and the type-converting equality operator is neither reflexive nor transitive:

Strict Equality Operator

- Not reflexive, because `NaN === NaN` is false.

Type-converting Equality Operator

- Not reflexive, because `NaN == NaN` is false.
- Not transitive, because `"1" == 1` and `1 == "01"` does not imply `"1" == "01"`.

Moreover, comparing values (by using their variable names) for equality needs to be considered carefully. For example, consider the following code snippet that compares the values of two variables `a` and `b`.

```
1 if(a === b) {
2   ...
3 }
```

Listing 13.1 Example of an if-condition in JavaScript.

The `if` condition tests if both variables, `a` and `b` evaluate to an equal value. If yes, the condition's body is evaluated. One can assume that we are free to use either `a` or `b` in the condition's body. However, JavaScript's `with (head) { ... body ... }` statement can be used to modify the current environment by placing `head` on top of the scope chain while executing `body`. Variable access that matches a property defined in `head` refers to the object's property. Because the `head` object could be a proxy (or an object that contains a getter function), the `head` object may influence the return arbitrarily.

With this construction, which is somewhat related to dynamic binding [52], we can dynamically change bindings in the scope chain by modifying a property defined in `head`. So, the first access to a variable `a` and `b` could return the same value for both variables, whereas the second access could return something different.

In JavaScript, guarantees are only given for values, i.e., for expressions that were evaluated to values before applying the equality operator, or for variables that are defined in the same

[1] This should not be confused with JavaScript's type converting equality operation that first converts both operands to a value of the same type.

scope. Moreover, primitive values require to use the strict equality operator to give any kind of guarantees.

To sum up, most JavaScript developers use equality to test a value to be `null` or `undefined` or to compare primitive values. Thus, programmers expect an equality operator to be an equivalence relation.

Treating proxies as transparent does not violate this expectation. It only weakens the assumption that pointer-equal values imply observational equivalence. To overcome this, transparent proxies could be restricted to projections that either behave identically to the target object or that throw an exception.

13.3 Alternative Designs

As we have seen in Section 12.2, the expected result of comparing proxies depends on the use case. This section explores some design alternatives for proxy equality and discusses their usability.

13.3.1 Program Rewriting

The simplest way to obtain transparency is to provide proxy-aware equality functions, for example `Proxy.isEqual()` and `Proxy.isIdentical()`, and to replace all uses of `==` and `===` with their proxy-aware counterparts. An internal map can be used to store the target object for each proxy object proxy, and a recursive lookup can reveal the base target for each proxy. These functions can be implemented in JavaScript, and the proxy constructor could be extended to maintain this map.

This approach enables to treat some proxies as transparent, whereas all other proxies remain opaque, and it allows to distinguish transparent proxies from their target objects in the implementation of the proxy abstraction.

However, it requires to transform the program before its execution. A macro system, for example, Sweet.JS for JavaScript [101], can elegantly do this. Unfortunately, Sweet.JS is implemented as an offline source code transformation, and it is not able to deal with JavaScript's dynamic features like `eval` and `with`.

13.3.2 Additional Equality Operators

Another approach is to reimplement JavaScript's equality operators `==` and `===` to unroll proxy objects and to compare the proxy's target objects. Two new operators, e.g., `:==:` and `:===:`, can be used as their opaque cousins to preserve the behavior of the current implementations of `==` and `===`.

`==` and `===` are supposed to be used in application code, whereas the implementation of a proxy library could use `:==:` and `:===:` to distinguish proxy and target objects.

This approach does not need any source code transformation. However, it is not clear how to ensure that application code does not use the opaque operators. Given both operations, the application code can test for proxies:

```
1 ((objectA==objectB) != (objectA:==:objectB));
```

Listing 13.2 Equality test for proxy objects.

This equality test returns true if one object is a proxy for the other object, and otherwise false.

13.3.3 Trapping the Equality Operation

Instead of introducing new operators, another approach is to trap the equality of a proxy object. A proxy handler could be extended by an optional boolean trap that decides whether the proxy is transparent or not when used in an equality test.

```
isTransparent : function () → boolean
```

Listing 13.3 Signature of an equality trap.

If the handler's trap returns a `false` value (according to JavaScript's definition of falsy values) or if it is not present, then the associated proxy behaves opaquely. Otherwise, it behaves transparently. Furthermore, the implementation of a proxy library can flip the transparency of a proxy object by reconfiguring the handler in the library code, analogous to the implementation of revocable references [20].

The following example shows the implementation of a function `wrap` that flips the transparency. The corresponding flag is hidden in the function closure and cannot be manipulated from the outside. The function makes proxies opaque before it checks the presence of the target value in a map.

```
const wrap = (function() {
  let flag = true;

  return function (target) {
    flag = false; // make proxies opaque
    if(!map.has(target)) { // create new proxy, if not existing
      let handler = {isTransparent:function() {
        return flag;
      }};
      map.set(target, new Proxy(target, handler));
    }
    let proxy = map.get(target);
    flag = true; // make proxies transparent before return
    return proxy;
  }
})();
```

Listing 13.4 Implementation of a `wrap` function which flips the transparency.

This design enables all scenarios in Section 12.2. However, it violates the assumption that object equality is an equivalence relation as the trap may randomly return another value.

13.3.4 Transparent Proxies in the VM

Yet another approach is to make proxies generally transparent. This design guarantees that object equality remains an equivalence relation, but it makes it impossible to test whether a reference is a proxy or a non-proxy object.

Unfortunately, there are use cases that must be able to distinguish proxy from non-proxy objects, for example, to improve efficiency. Thus, for implementing proxy abstractions, it must be possible to break the transparency.

This could be done by using object-capabilities. A capability (this could be an object or a message) describes a transferable right to perform a particular operation. Such a capability object can easily be given to the constructor of entirely transparent proxies. The capability can be hidden in the scope of the function that wraps objects.

```
const wrap = (function() {
  const capability = {};
  return function(target, handler) {
    return new TransparentProxy(target, handler, capability);
  };
})();
```

Listing 13.5 Implementation of a `wrap` function using a capability.

Later, this capability can be used to see the real identity of the proxy's id, e.g., to check the presence of a target object in a weak map. The following snippet demonstrates the outcome.

```
Object.equals(object, proxyA, capability); // returns true
Object.equals(proxyA, proxyB, capability); // returns true
Object.equals(object, proxyA, { /* some other object */ }); // returns false
Object.equals(proxyA, proxyB, { /* some other object */ }); // returns false
```

Listing 13.6 Outcome of a transparent proxy.

WeakMap's and other internal data structures using object equality can implement this object-capability model in the same way.

14 Transparent Proxies

When comparing two objects for equality, a *transparent proxy* is (recursively) replaced by its target object. Its suggested use is to implement projections, e.g., projection contracts: the proxy either returns a value identical to the value that would be returned from the target object (this also includes the same side effects) or it throws an exception. Thus, contracts become invisible until a contract is violated.

This chapter presents a prototype implementation of transparent proxies in the VM and reports on performance tests that were taken from evaluating JavaScript programs.

14.1 The User Level

As opaque and transparent proxies are intended for different use cases, we do not remove or replace the existing proxy implementation. A new proxy constructor `TransparentProxy` can be used to create a transparent proxy. Like the standard `Proxy` constructor, the new `TransparentProxy` constructor takes two arguments: a target object, which might be a non-proxy object or another (transparent or opaque) proxy object and a handler object that may contain the same optional trap functions.

```
1 let proxy = new TransparentProxy(target, handler);
2 proxy === target // evaluates to true
```

Listing 14.1 Just a new Proxy Constructor.

Here, `target` and `handler` are arbitrary JavaScript objects. The newly created transparent proxy `proxy` is transparent with respect to object equality. Whenever the transparent proxy is used in an operation that builds on object equality, then it returns the identity of its target object. In case that the proxy's target object is another transparent proxy, then unrolling proceeds on the proxy's target object until it reaches a non-proxy object or an opaque proxy.

Thus, comparing `proxy` with its target object returns `true`, instead of `false`.

```
1 proxy === target // evaluates to true
```

Listing 14.2 Comparing proxy and target object.

Furthermore, a transparent proxy is equal to all other transparent proxies of the same target and to all chains of nested proxies with the same base target.

```
1 let proxy2 = new TransparentProxy(target, handler);
2 proxy === proxy2 // evaluates to true
```

Listing 14.3 Comparing nested proxy objects.

```
1 let proxy3 = new TransparentProxy(proxy2, handler);
2 proxy3 === target // evaluates to true
3 proxy3 === proxy // evaluates to true
```

Listing 14.4 Comparing nested proxy objects (cont'd).

In addition to the comparison operators (`==`, `!=`,`===`, and `!==`), transparent proxies must be unrolled in all situations. For example when using them as a key value in a keyed collection like maps or sets or when matching switch clauses.

A keyed collection is a data structure that uses values as key, for examples to map a value to another value in a map object. As these collections use value (object) equality on

their key values, we must unwrap a transparent proxy to resolve a correct key value for this proxy object. Our approach guarantees that identical objects map to the same field in a keyed collection. The following example shows its behavior.

```
1 let set = new Set();
2 map.add(target);
3 map.add(proxy);
4 map.size; // returns 1
5 map.has(proxy2); // returns true
6 map.delete(proxy3); // returns true (successfully removed)
```

Listing 14.5 Using a transparent proxy as key value.

But, there is a peculiarity: When adding `target` in line 2 to the set its internal entries list gets extended by the new entry. As in line 3 the proxy object is already part of the list is not updated. Thus, when iterating over all elements, we still get `target`, the first added element, and not the last one. This behavior also exists in JavaScript maps.

Furthermore, a `switch` statement need to unwrap a transparent proxy that is used as an expression value or as a case value before matching the case clauses.

```
1 let proxy = new TransparentProxy(target, handler);
2 switch (proxy) {
3   // some other case clauses
4   case target:
5     // do something
6     break;
7 }
```

Listing 14.6 Matching a switch clause with a transparent proxy.

To sum up, transparent proxies are transparent with respect to object equality in all situations. This design enables all the use cases in Section 12.2 and it preserves JavaScript's guarantee that object equality is an equivalence relation.

However, transparent proxies are slippery. Making them generally transparent makes it impossible to test whether a reference is a proxy or an original object. As there are abstractions that require to reveal the identity of a proxy library code may want to break the transparency of a proxy. For example, the implementation of access permissions contracts [65] extracts the current permission from a proxy to construct a new proxy with updated permission. This improves the efficiency of the implementation, which would otherwise generate long chains of proxy objects. But, with this implementation, it is hard to manipulate because transparent proxies have no own (visible) identity.

14.1.1 Identity Realms

For implementing proxy abstractions, it must be possible to reveal the real identity of a transparent proxy. Thus, we use object-capabilities to create proxies in a particular *identity realm* and to create an equality function that reveals proxies of that realm. A capability describes a transferable right to perform a specific operation, for example, to break the transparency of a proxy object.

The realm constructor is implemented on top of the transparent proxy implementation and creates a new identity realm, represented by a JavaScript object.

```
1 var realm = TransparentProxy.createRealm();
```

Listing 14.7 Creating a new identity realm.

The realm object consists of a proxy constructor (named `Proxy`), an `equals` and an `identical` function (which are related to JavaScript's comparison operators *equal* and *strict equal*), and a new constructor for all keyed collections, namely `Map`, `Set`, `WeakMap`, and `WeakSet`.

```
var proxy = realm.Proxy (target, handler);
```

Listing 14.8 Creating a transparent proxy in an identity realm.

The `Proxy` constructor creates a new transparent proxy of that realm. As before, the proxy is transparent unless someone uses `realm.equals` or `realm.identical`. Here, `realm.equals` is a capability that represents the right to reveal the real identity of proxies from that realm.

```
proxy === target; // evaluates to true
realm.equals(proxy, target); // evaluates to false
```

Listing 14.9 Comparing transparent proxies in a realm.

The functions `realm.equals` and `realm.identical` are not restricted to proxy object exclusively. They adopt the behavior from JavaScript's equals (`==`) and strict equals (`===`) operator such that they can be used instead of the built-in comparison operators. Even though both functions behave identical for objects, we may also apply the functions to primitive values. Therefore, we need both variants as both behave differently for primitive values.

14.1.2 Realm-aware Data Structures

In addition to realm-aware equality functions, the realm object also provides new constructors for all kinds of keyed collection. Instances of those collections see the real identity of proxies of that realm. As before, it follows the principle that identical objects (in that identity realm) have to point to the identical entry (in realm-aware data structures). Two transparent proxies of the same realm are not identical; therefore they have to address different fields in a collection. The following example demonstrates this behavior.

```
var realm = TransparentProxy.createRealm();
var proxy = realm.Proxy(target, handler);
var map = realm.Map();
map.add(proxy, 1); // map : [#proxy → (proxy, 1)]
map.add(target, 2); // map : [..., #target → (target, 2)]
map.size; // returns 2
```

Listing 14.10 Using realm-aware keyed collections.

As `proxy` and `target` are not identical with respect to the identity realm `realm` they address different fields in the realm-aware map `map`. However, for every other map they remain transparent and map to the same field.

Weak maps and other internal data structures are implemented in the same way.

14.2 Implementation

We implemented a prototype extension of the SpiderMonkey JavaScript engine [99], which provides a new transparent proxy constructor `TransparentProxy` according to the design in Section 14.1. SpiderMonkey is Mozilla's JavaScript engine for Gecko and used in various Mozilla products like Firefox. The SpiderMonkey engine contains an interpreter, two just-in-time compilers (a baseline compiler and the IonMonkey optimizing compiler), and a garbage collector. The baseline compiler is a warm-up compiler for IonMonkey and brings light

performance improvements. The IonMonkey compiler brings high-performance optimizations and enables to reduce the memory usage of SpiderMonkey.

Changing the equality leads to a couple of adjustments to the interpreter, the baseline compiler, and the IonMonkey optimizing compiler.

14.2.1 JavaScript Interpreter

To support transparent proxies the interpreter has to be revised in several places:

1. The interpreter has to be extended by a new `TransparentProxy` object.
2. All comparison operators have to be modified to unroll transparent proxies to obtain the identity object from that proxy.
3. All keyed collections that are connected to object equality have to be adjusted.

14.2.1.1 The `TransparentProxy` Object

The new `TransparentProxy` object extends the set of existing proxy implementations by a new kind of proxy. It inherits all features from its opaque cousin and can be used in the same way. The only difference is its transparency. No modifications were done to the existing (opaque) proxy.

14.2.1.2 JavaScript's Equality Comparison

JavaScript provides two kinds of comparison operators. The strict equality operator (e.g., `===`) returns `false` if both operands are not of the same type, whereas the type converting equality operator (e.g., `==`) first converts both elements to the same type before it applies the corresponding strict equality operator. For objects, both operands behave identically.

The interpreter's strict equality comparison `x===y`, called with values x and y, produces `true` or `false`. The comparison works as follows [33, 7.2.13]:

1. If *Type(x)* is different from *Type(y)*, return *false*.
2. If *Type(x)* is Undefined, return *true*.
3. If *Type(x)* is Null, return *true*.
4. If *Type(x)* is Number, then
   a. If x is *NaN*, return *false*.
   b. If y is *NaN*, return *false*.
   c. If x is the same Number value as y, return *true*.
   d. If x is $+0$ and y is -0, return *true*.
   e. If x is -0 and y is $+0$, return *true*.
   f. Return *false*.
5. If *Type(x)* is String, then
   a. If x and y are exactly the same sequence of code units (same length and same code units at corresponding indices), return true.
   b. Else, return *false*.
6. If *Type(x)* is Boolean, then
   a. If x and y are both true or both false, return *true*.
   b. Else, return *false*.
7. If x and y are the same Symbol value, return *true*.
8. If x and y are the same Object value, return *true*.

9. Return *false*.

The algorithm starts with checking if both values are of the same type. Then it handles all comparisons of primitive values before it applies pointer equality.

To cater for transparent proxies, the algorithm needs to unroll a transparent proxy to its identity object before applying pointer equality. The internal method *GetIdentityObject* returns the object that should be used in an equality test. Our implementation extends the algorithm as follows:

1. ...
8. If x is Object, then
   a. Let *lhs* be the result of calling *GetIdentityObject* on x.
   b. Let *rhs* be the result of calling *GetIdentityObject* on y.
   c. If *lhs* and *rhs* refer to the same object, return `true`.
   d. Otherwise, return `false`.
9. ...

14.2.1.3 Getting the Identity Object

When comparing two objects for equality, transparent proxies may not use their own identity. To get the right identity for an operation, all global object comparisons refer to an internal `GetIdentityObject` method, that returns the correct identity for that operation. The following listing shows a pseudo-code implementation of this algorithm.

```
function GetIdentityObject(object, realm) {
  if (isTransparentProxy(object) && object.realm!==realm) {
    return GetIdentityObject(getProxyTargetObject(object), realm);
  } else {
    return object;
  }
}
```

Listing 14.11 Pseudo-code for `GetIdentityObject`.

For a non-proxy object and an opaque proxy, the function returns its argument. For a transparent proxy object, `GetIdentityObject` returns the result from applying `GetIdentityObject` to the proxy's target object. The method stops the traversal when reaching a transparent proxy of a particular realm.

The following enumeration gives a brief scratch of the implementation used in the prototype implementation. Every transparent proxy carries an internal slot that contains its identity realm. When the *GetIdentityObject* internal method is called with argument O and realm R the following steps are taken:

1. Assert: *Type(O)* is Object.
2. If O is not a TransparentProxy object, then return O.
3. If R is not null, then
   a. Let *realm* be the value of the [[ProxyRealm]] internal slot of O.
   b. If *realm* equals R, return O.
   c. Else, let *target* be the value of the [[ProxyTarget]] internal slot of O.
   d. Return the result of calling *GetIdentityObject*, passing *target* as the argument and R as realm.
4. Else, let *target* be the value of the [[ProxyTarget]] internal slot of O.

5. Return the result of calling *GetIdentityObject*, passing *target* as the argument and *null* as realm.

14.2.1.4 Realm-aware Equality Comparison

Completely transparent proxies are indistinguishable from their base target. However, to make them distinguishable, `realm.equals` and `realm.identical` can be used to detect a distinction.

When `realm.equals` is called with arguments `x`, `y` the following steps are taken:

1. If x is not a TransparentProxy object or y is not a TransparentProxy object, then
   a. Return the result of applying the built-in equality method to x and y.
2. Else
   a. Let *realm* be the value of the [[Realm]] internal slot of *realm.equals*.
   b. Let *lhs* be the result of calling *GetIdentityObject*, passing x as an argument and *realm* as realm.
   c. Let *rhs* be the result of calling *GetIdentityObject*, passing y as an argument and *realm* as realm.
   d. Let *result* be the value of testing the equality of the *lhs* and *rhs* references.
   e. Return the result of *ToBoolean*(*result*).

14.2.1.5 Realm-aware Keyed Collections

In addition to the comparison operators, all internal data structures that depend on object equality (e.g., `Map`, `Set`, and `WeakMap`) need to be adjusted to handle transparent proxies. Whenever `target == proxy` in a particular identity realm, then `target` and `proxy` must address the same field in a collection in that realm.

When adding a new element to a set (or a key-value pair in a map), the operation first calls the `GetRealmIdentityObject` internal method to determine the identity object of that key value and computes a hash value for the identity object. However, the original key value is stored in the collection, not its identity object. This guarantees that maps and sets cannot be used to unwrap proxies and that iterating over the keys still returns the proxy object.

14.2.2 Baseline Compiler

The SpiderMonkey Baseline Compiler is the first tier of the SpiderMonkey JIT compilation process. It collects information by using inline caches for some operations and it produces native code for JavaScript through stub method calls.

To adjust the Baseline Compiler, the fallback stub was modified to do a VM call when comparing two objects for equality and to compare both objects in exactly the same manner as done in the interpreter. This step is required because the Baseline Compiler dies not see the base target for a transparent proxy. Hence, it is not possible to optimize object comparisons involving transparent proxies.

14.2.3 IonMonkey Compiler

The IonMonkey optimizing compiler is the final tier of the SpiderMonkey JIT compilation process. It runs on top of the Baseline Compiler. Like before, the IonMonkey optimizing compiler did not see the base target for a transparent proxy.

When comparing two objects for equality, we stop emitting optimized code for any object-object comparison that involves a `TransparentProxy` object. We use the fallback stub to do a VM call and to do a comparison with the identity object. Any other kind of comparison remains unaffected and still happens through the optimized stubs.

14.2.4 Getting the Source Code

The implementation of the modified engine is available on the Web[1]. Its branch *global-tproxy-object*[2] contains the implementation of the `TransparentProxy` object and all modifications shown in this chapter. A *README.txt* file in the branch links to the build instructions.

14.3 Performance

This section reports on the performance of our modified engine. In detail, it considers the question if the introduction of transparent proxies affects the performance of the JavaScript code.

In a first experiment, we applied our modified engine to JavaScript benchmark programs that do not use proxies at all to evaluate the performance impact on non-proxy code. However, these programs are affected by our changes to the equality comparison algorithm that is used by all built-in equality operators as well as in maps and sets.

In a second experiment, we add (opaque and transparent) proxies to the benchmark programs to evaluate the impact on code that makes use of proxies.

14.3.1 The Testing Procedure

Again, we use benchmark programs from the Google Octane 2.0 Benchmark Suite [85].

In our first experiment we simply run the benchmark programs in our modified engine, and in a baseline engine to obtain comparative values. In both cases, we run the benchmarks in different configurations to distinguish the impact on the interpreter from the impact on the baseline compiler and the IonMonkey compiler.

In our second experiment, we apply a source-to-source compiler to the benchmark program to wrap all function expressions in a proxy object, which, when applied to some values, recursively implements a proxy membrane on that values.

All benchmarks were run on a benchmarking machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory.

14.3.2 Results

Table 14.1 contains the score values of all benchmark programs in different configurations and Table 14.2 shows the percentage variance between the score values from the unmodified engine and the score values from the modified engine. All scorings were taken from a deterministic run and by using a warm-up run.

Comparing the total scores of the interpreter (sub-column **No-JIT**), the modified engine is 0.55% slower than the unmodified engine. When comparing the total scores of the baseline compiler (sub-column **No-Ion**) we see that the modified engine is 0.21% slower than the

[1] `https://github.com/keil/gecko-dev`
[2] `https://github.com/keil/gecko-dev/tree/global-tproxy-object`

Benchmark	Origin			Modified		
	Full	**No-Ion**	**No-JIT**	**Full**	**No-Ion**	**No-JIT**
Richards	14368	403	62	14245	430	64
DeltaBlue	15657	403	73	16210	393	74
Crypto	13143	794	119	13018	791	119
RayTrace	37739	428	156	37788	408	156
EarleyBoyer	11726	930	247	11891	898	250
RegExp	1554	755	336	1538	748	320
Splay	9663	1211	500	9874	1213	515
SplayLatency	14205	4216	2762	12980	4396	2723
NavierStokes	15875	1016	180	15905	1011	179
pdf.js	5623	2801	719	5643	2858	708
Mandreel	8313	433	86	8339	429	86
MandreelLatency	8045	2730	547	8055	2680	514
Gameboy Emulator	16738	3115	536	16479	3163	523
Code loading	9128	9304	9971	9136	9333	9938
Box2DWeb	14977	1631	317	14564	1609	312
zlib	36945	36909	36746	36897	36945	36685
TypeScript	13921	3190	1207	13733	3111	1217
Total Score	11904	1620	484	11840	1617	481

Table 14.1 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **Origin** gives the baseline scores for the unmodified engine and column **Modified** contains the scores for running with the modified engine. The sub-column **Full** lists the scores with enabled IonMonkey compiler, whereas sub-column **No-Ion** *(no IonMonkey)* lists the scores with enabled baseline compiler, but disabled IonMonkey compiler. The last sub-column **No-JIT** *(no just-in-time compilation)* shows the scores without any kind of just-in-time compilation.

unmodified engine and comparing total scores of the IonMonkey compiler (sub-column **full**) we see that the modified engine is 0.54% slower than the unmodified engine.

However, when looking to the score values of particular benchmark programs we see that some benchmarks are faster in the modified engine than in the unmodified engine. For example, the DeltaBlue benchmark runs 1.22% faster in the interpreter mode, 2.72% slower in the Baseline compiler mode, and 3.41% faster in the IonMonkey mode. But, all observed differences are in the range of the standard deviations of the mean score produced by the unmodified engine. When looking at the total score, we got a standard deviation for the total score of 2 score points in the interpreter mode, a difference of 13 score points in the baseline compiler mode, and a standard deviation of 62 score points in the IonMonkey mode.

The numbers indicate that our modifications do not have an observable impact on the execution time of non-proxy code. The reason is that the majority of all equality tests are between primitive values or test an object for null or undefined. However, our modification only applies when both operands are objects.

14.3.3 Threats to Validity

A reader might argue that the subject programs are uninteresting because they do not contain proxies. The benchmarks are taken to measure the impact on the execution of non-proxy code. Clearly, long chains of nested transparent proxies will increase the execution time.

To make a valid comparison with the original engine, we have to take code that evaluates to the same outcome in all engines. The original engine makes it impossible to replace objects

Benchmark	Percentage Variance			Standard Deviations		
	Full	No-Ion	No-JIT	Full	No-Ion	No-JIT
Richards	0.86%	-6.28%	-2.20%	45	14	0.36
DeltaBlue	-3.41%	2.72%	-1.22%	776	2	0.40
Crypto	0.96%	0.34%	0.00%	73	5	0.44
RayTrace	-0.13%	4.73%	-0.21%	740	2	0.44
EarleyBoyer	-1.38%	3.56%	-1.07%	190	7	0.67
RegExp	1.02%	0.94%	4.89%	10	8	2.67
Splay	-2.14%	-0.11%	-2.98%	360	64	31.11
SplayLatency	9.44%	-4.09%	1.43%	378	542	84.44
NavierStokes	-0.19%	0.49%	0.56%	16	4	0.67
pdf.js	-0.34%	-1.98%	1.60%	289	76	1.78
Mandreel	-0.30%	0.93%	0.08%	9	22	0.56
MandreelLatency	-0.12%	1.88%	6.49%	511	52	2.89
Gameboy Emulator	1.57%	-1.50%	2.42%	1787	36	3.33
Code loading	-0.08%	-0.31%	0.34%	92	30	18.00
Box2DWeb	2.84%	1.35%	1.82%	16	7	0.44
zlib	0.13%	-0.10%	0.17%	40	111	45.33
TypeScript	1.37%	2.53%	-0.82%	333	13	1.33
Total Score	0.54%	0.21%	0.55%	62	13	2.00

Table 14.2 Percentage variance and Standard Deviation for the Google Octane 2.0 Benchmark Suite. Column **Percentage Variance** shows the percentage variance between the score values from the unmodified engine and the score values from the modified engine. Column **Standard Deviations** shows the standard deviation of the mean score produced by the unmodified engine.

with their (opaque) proxies without influencing the execution. As reported in Chapter 12 some object-object comparisons flip their result after introducing proxies.

15 Observer Proxy

The presented transparent proxy (Chapter 14) is a straightforward extension of the already existing opaque proxy. However, the intended use cases of a transparent proxy (the implementation of projection contracts) do not require that the transparent proxy is as powerful as the opaque proxy. Quick recap: the existing proxy implementation allows to redefine the semantics of the underlying target object in many aspects.

For implementing a transparent contract wrapper, it would be sufficient to restrict transparent proxies to projections: i.e., it either behaves identical to the target object, or it restricts the behavior of the target object, e.g., by throwing an exception.

A so-called *Observer Proxy* has to produce the same side effects as the target object, and it has to return a value identical to the value that would be returned from the target object, or it has to throw an exception. Returning an identical value includes the return of an Observer of this value to implement a membrane and to further restricts the behavior. Such an Observer can cause a program to fail more often, but in case it returns it behaves in the same way as if no observers would be present.

A similar feature is provided by Racket's chaperone [100] proxy. A chaperone either returns an identical value, a chaperone of this value, or throws an exception.

15.1 Draft Implementation of an Observer Proxy

This section sketches the implementation of an observer proxy in JavaScript. The implementation is based on a transparent proxy as presented in Chapter 14 and adapts the semantics of Racket's chaperone proxy [100] in some aspects. Its design obeys the following rationales:

Interference-free Monitoring User-defined traps are not allowed to perform effects other than the target object would do. This means that traps can inspect the operation's arguments and the operation's return as long as they do not attempt to cause observable effects. However, in JavaScript, each property access might be the call of a side-effecting getter function. This requires that the observer protects the arguments from any write operation, which also includes a prohibition of getter functions.

Projection After inspection, the user-defined traps must either throw an exception or perform equal to the default operation. Also, each trap can implement further restrictions on the given arguments by wrapping the values in an observer, e.g., to implement a membrane or to assert domain contracts to the function arguments.

Permitted Effects User-defined traps must not cause side effects on the given arguments. However, the implementation of contracts and membranes may require to store data (e.g., the current evaluation state) for later use, to remember already wrapped objects to avoid re-wrapping, or even to call external functions and constructors. This requires that traps are permitted to perform effects on a certain domain.

The implementation is available as a JavaScript library on the web[1].

[1] `https://github.com/keil/Observer.js`

```
1 var handler = {
2   get: function(target, name, receiver, continue) {
3     continue(target, name, receiver, function inspect(result, commit) {
4       commit(result);
5     });
6   }
7 }
8 var target = { /* some object */ };
9 var proxy = new Observer(target, handler);
```

Listing 15.1 Draft implementation of a handler object for an observer proxies.

15.1.1 The User Level

Listing 15.1 demonstrates the implementation of a handler object for an observer proxy. The example shows the implementation of the `get` trap. Other traps can be implemented in the same way.

The constructor `Observer` (line 9) creates a new observer proxy. Like opaque proxies, the constructor consumes two parameters: a `target` object, which can be a native or another proxy object, and a `handler` object containing traps to observe operations on `proxy`. Like before, the handler is a placeholder object for optional trap functions.

Observer proxies extend the already existing opaque (transparent) proxies and provide an API similar to the API of opaque proxies, i.e., they mediate the same operations, and they look for the same trap functions in the handler object. However, traps are only allowed to inspect the operation or to implement further restrictions.

To this end, all values that are given to the trap are wrapped in a proxy membrane that protects the values from unintended modifications. As the trap is not able to perform the usual operation (e.g., a property set) observer traps apply a continuation-passing style (CPS) to enable inspection and to continue operations.

All traps are called before the operation performs. This enables to inspect the operation's arguments (e.g., the property to get or the new value of a property), to implement further restrictions by wrapping the values in another observer, or to throw an exception. After inspection, the trap continues with the default operation by calling a continuation function (`continue`), which is given as the last argument to the trap function.

For example the `get` method (line 2), which is a trap for handling property access. Its parameters are the target object, the property name to get, and the receiver object. In addition, it takes a `continue` function to continue the original operation. Performing an operation like `proxy.x` calls the `get`. This enables to inspect the arguments or to implement further restrictions on the parameters. To continue with the property lookup, the trap calls this `continue` function with the traps parameters. Also, `continue` consumes another continuation function (here called `inspect`) to continue inspection after doing the property lookup and before return the value to the user.

Internally, `continue` first checks if the arguments are identical to the values given to `get`. Then it proceeds to call the standard operation on the target value. After doing this, it calls `inspect` with the operations return to continue inspection and another continuation function, `commit` to return a value. Again, the return value gets wrapped in a proxy membrane to protect the value from modifications.

Figure 15.2 shows the implementation of a `wrap` function implementing a proxy membrane based on observer proxies. The observer checks if a property exists before getting a property

```
function wrap(target) {

  if(!(target instanceof Object))
    return target;

  var handler = {
    get: function(target, name, receiver, continue) {
      // checks for undefined property names
      if(!(name in target))
        throw new Error(`Access to undefined property ${name}.`);
      // continue property lookup
      continue(target, name, receiver, function inspect(result, callback) {
        // implement membrane
        callback(wrap(result));
      });
    }
  }
  return new Observer(target, handler);
}
```

Listing 15.2 Implementation of a proxy membrane using observers.

value. If the property exists it continues with the property lookup and wraps the return value in the same membrane. Otherwise, it throws an exception.

15.1.2 Under the Hood: The Meta-Handler

The implementation of observer proxies uses *meta-level funneling* [20] to mediate access to the user-defined handler object. Meta-level funneling is a technique that implements another proxy as the base-proxy's handler object.

in general, the implementations distinguish between a user-defined handler (called `handler`) that contains traps for observing operations on the proxy object (cf. Section 15.1.1) and an internal meta-handler (called `controller`) that implements the behavior of the observer.

Figure 15.1 illustrates the implementation of a meta-handler. Performing an operation (like property get or property set) on the observer object results in a meta-level call on the proxy's handler object, i.e., a property get on the proxy's handler to get the corresponding trap. As this object is again a proxy object, the request for trap ends up in a meta-level call on the controller object by calling the get-trap on the controller object. This trap now devices what to do with the current operation: it may forward the operation to the user-defined trap or it may trigger the default operation. In our case, the controller calls the `calltrap` methods, which handles all trap operations of the observer proxy.

Figure 15.3 shows the implementation of the constructor function for observer proxies. The constructor function takes the target object and the user-defined handler. In addition, there is a flag that instructs the constructor to store the observer in a map for later use. This is required to reveal observers, if necessary.

In line 5 the constructor first checks if the target object is a sandbox proxy. This step is required because observer traps never see the unprotected values. They always see sandbox proxies. However, for implementing a membrane, it is necessary to wrap values in another observer. Thus, instead of implementing the observer on top of the sandbox wrapper, the observer gets implemented on the unwrapped sandbox value, and the sandbox proxy gets

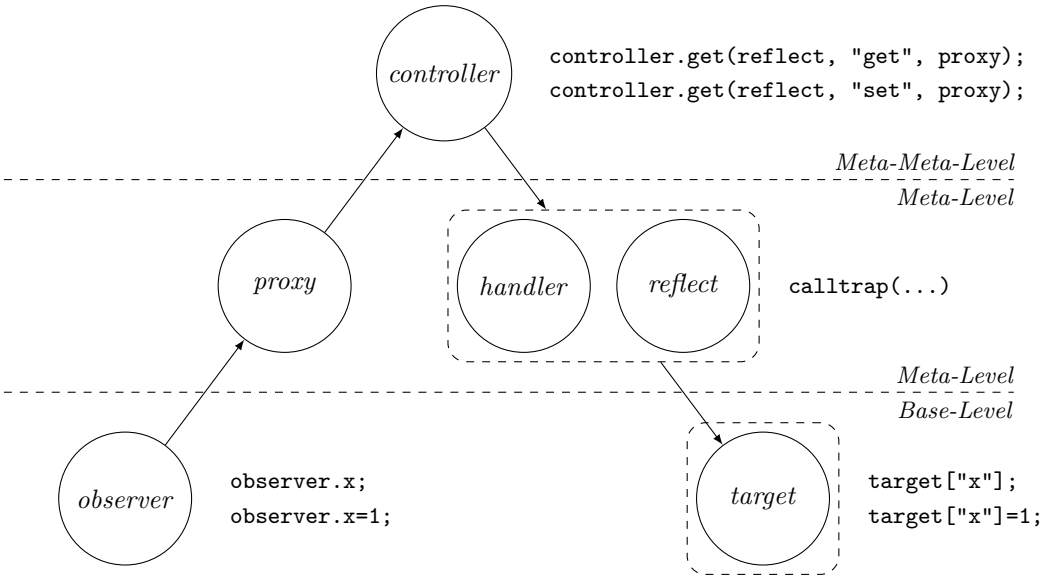


Figure 15.1 Example of an observer operation. The property get `observer.x` end up in an trap request on the controller object by calling the trap `controller.get(reflect, "get", proxy);` and the property set operation `observer.x=1` ends in an trap request on the controller by calling the trap `controller.get(reflect, "set", proxy);`.

re-implemented on top of the observer. The other construction steps are straightforward: It first gets the Proxy constructor from its identity realm before it wraps the target object in a new transparent proxy.

Figure 15.2 illustrates the implementation of the proxy-chain inside of a handler object while implementing a proxy membrane through observer proxies. In the first part, we have a sandbox proxy wrapping a target object. When wrapping the sandbox-proxy in another observer proxy, the constructor installs the proxy below of the sandbox proxy, as the second part demonstrates. In the third part, we have the situation after installing a second observer proxy on an already wrapped object.

Listing 15.4 shows the implementation of the controller. It is a simple proxy handler that defines the `get` trap. As the controller is a meta-handler, it's trap gets called whenever looking for a trap in the user-defined handler.

The trap first checks if the requested trap is defined. If yes, it returns a function which handles calling the user-defined trap. Otherwise, it returns the default method from `Reflect`.

Listing 15.5 shows the implementation of function `calltrap`. The function is responsible for handling the inspection process of the user-defined trap. As arguments, it takes the user-defined trap `trap`, a function for the corresponding default operation, and a list of trap arguments as provided by the proxy objects for that trap.

It starts with applying the user-defined trap `trap` to the sandboxed `this` objects and the sandboxed arguments values. The provided continuation function first checks if the arguments provided by the user are identical to the default values. This comparison only works with transparent proxies. The user-defined trap needs to return a value identical to the default return, i.e., either the same value or an observer of that value.

Later, it performs the default operation and calls the user-defined post-operation-inspection with the sandboxed return value. It also provides another continuation function, `commit`, to complete the operation. As before, function `commit` first checks if the return value

```
function Observer(target, handler, keep=true) {
  if(!(this instanceof Observer)) return new Observer(target, handler, keep);

  // wrapping of a sandbox proxy.
  if(sandbox.has(target)) {
    return sandbox.wrap(new Observer(sandbox.unwrap(target), handler, keep));
  }

  // Proxy Constructor
  const Proxy = realm.Proxy;

  // create new observer based on the given  handler
  const observer = new Proxy(target, new Proxy(Reflect, new Controller(handler)));

  // remembers existing observers
  if(keep) observers.add(proxy);

  // return new observer proxy
  return proxy;
}
```

Listing 15.3 Implementation of the `Observer` constructor.

```
function Controller(handler) {
  if(!(this instanceof Controller)) return new Controller(handler);

  this.get = function(Reflect, trapname, receiver) {
    return (trapname in handler) ? function () {
      return calltrap(handler[trapname], Reflect[trapname], Array.from(arguments))
    } : Reflect[trapname];

  }
}
```

Listing 15.4 Implementation of the `Controller` handler.

```
function calltrap(trap, default, argumentsList) {

  // Default trap return.
  let result = undefined;

  trap.call(sandbox.wrap(this), ...sandbox.wrap(argumentsList), function(...args) {

    // Function inspect
    const inspect = args.pop();

    if((typeof inspect) !== 'function') throw new TypeError();

    // Unwrap and check arguments.
    for(var i in args) {
      var arg = sandbox.unwrap(args[i]);

      if(arg === argumentsList[i]) argumentsList[i] = arg;
      else throw new ObserverError("Argument values must be equal to the default arguments.");
    }

    // Call the default operation.
    result = default.apply({}, argumentsList);

    // continues inspection
    inspect.call(null, sandbox.wrap(result), function commit(value) {

      // unwrap the return value
      let value = sandbox.unwrap(value);

      // check the return value
      if(result === value) result = value;
      else throw new ObserverError("Return values must be equal to the default return.");

    });
  });

  // Return result
  return result;

}
```

Listing 15.5 Implementation of function `calltrap`.

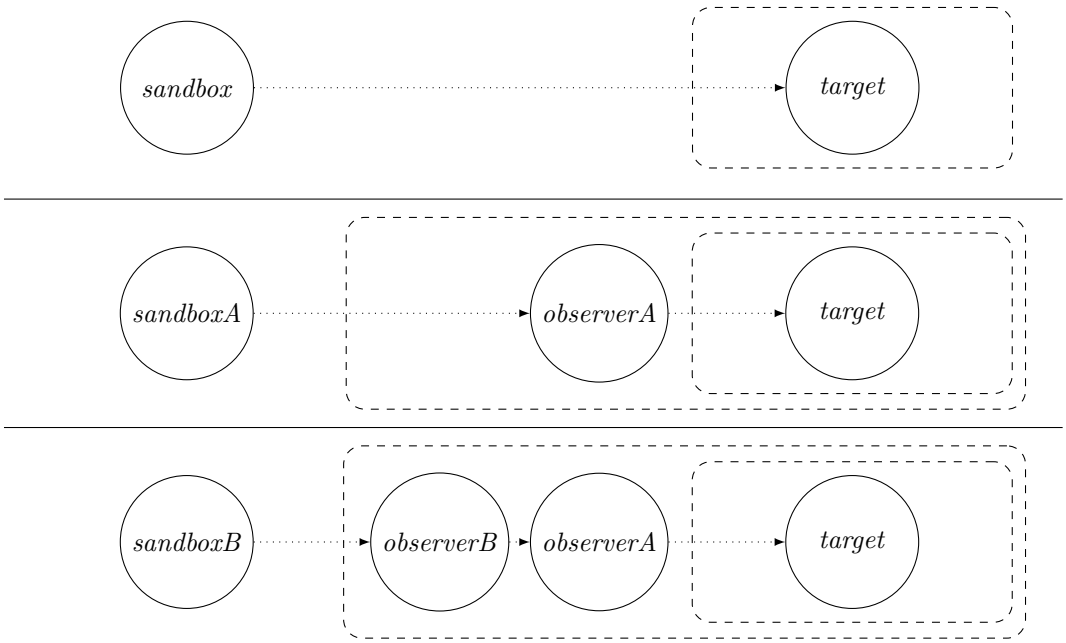


Figure 15.2 Example of implementing an observer membrane. The example demonstrates the proxy-chain inside of a handler object while implementing a proxy membrane through observer proxies.

returned from the user-defined trap is identical to the default return value. If yes it returns the value. Otherwise, it throws an exception.

15.1.3 Notes

JavaScript proxies are a powerful mechanism to redefine the semantics of an underlying target object. However, this power also has a cost. Proxies prevent the optimizing compiler from optimizing a program efficiently. Furthermore, as the semantics of the proxy object may differ from the semantics of its target, it is sensible to spend proxies an own identity.

However, this prevents some useful use-cases. Contract proxies, for example, did not need the full power as provided by JavaScript's built-in proxy objects. For implementing projection contracts it would be sufficient to restrict proxies to transparent observers.

Built-in observer proxies would also enable the JIT compiler to optimize a program more efficiently. Proxies are no longer able to perform arbitrary side effects or to return an arbitrary value.

IV

Static Contract Simplification

16 An Evaluation of Contract Monitoring

Dynamic contract monitoring has become a prominent mechanism while providing strong guarantees and still preserving the flexibility and expressiveness of a language. Writing formal and precise specifications in the form of contracts sounds appealing, but it comes with a cost: Dynamic contract monitoring degrades the execution time of the underlying program [102].

The costs arise because every contract extends a program with additional code that checks the contract while the program executes. Moreover, efficiency unconscious human developers may add contracts to error-prone functions and objects so that contracts end up on frequently used functions or hot-paths in a program. In particular, predicates may repeatedly check the same values, and different predicates may check redundant parts.

Existing contract systems like Racket's contract framework [45, Chapter 7], Disney's JavaScript contract system *contracts.js* [30], JSConTest2 [64], or TreatJS for JavaScript report a considerable runtime impact when extending programs with contracts.

In contrast, static contract checking [113] avoids additional runtime costs by removing contracts after inspection. However, static contract checking is not suitable for a language like JavaScript. The dynamic nature of JavaScript requires dynamic contract monitoring. Completely static techniques would lead to a huge number of false positives.

This part presents our ongoing work on *Static Contract Simplification* which attacks this issue with compile-time program transformation. Its goal is to apply static contract checking to simplify a contract and to obtain residual contracts that are collectively cheaper to check at runtime while preserving the original behavior of the program.

16.1 Motivation

Dynamic contract checking impacts the execution time of the underlying program. Source of this impact is (1) that every contract extends a program with additional contract code, (2) that the contract monitor itself causes some run-time overhead, and (3) that the presence of contracts prevents a program from being optimized efficiently. To illustrate this, we consider runtime values obtained from the TreatJS (cf. Chapter 7) contract framework for JavaScript.

TreatJS reports a heavy runtime deterioration because of the presence of intersection and union contracts require to continue contract checking on both operands, even if the contract monitor already observes a contract violation in a subcontract. The final violation depends on combinations of failures in different subcontracts. Moreover, the contract monitor must connect the outcome of each subcontract with the enclosing contract operation. Figure 16.1 contains the runtime values from running the Google Octane Benchmark Suite, and Figure 16.2 lists some numbers of internal counters (see also Section 7 for more details).

The numbers indicate that the heavily affected benchmarks (*Richards*, *DeltaBlue*, *RayTrace*) contain a very large number of predicate checks. For example, the *Richards* benchmark performs 24 top-level contract assertions (this are all unique contracts in a source program), 1.6 billion internal contract assertions (including top-level assertions, *delayed* contract checking, and predicate evaluation), and 936 million predicate checks. If we assume that every predicate check is at least one function call, one wrap operation on the subject value, and one update operation on the callback graph, then it does not come as a surprise that the overall run-time increases from 4 seconds to approximately 5 hours and 33 minutes.

Benchmark	Baseline	TreatJS	
	time (sec)	*time (sec)*	*slowdown*
Richards	4	19995	5093
DeltaBlue	3	28431	9285
Crypto	8	8	1
RayTrace	2	9510	4035
EarleyBoyer	56	PositiveBlame	
RegExp	6	6	1
Splay	3	162	55
SplayLatency	3	162	55
NavierStokes	4	4	1
pdf.js	6	27	4
Mandreel	5	PositiveBlame	
MandreelLatency	5	PositiveBlame	
Gameboy Emulator	4	1741	465
Code loading	9	9	1
Box2DWeb	4	2354	609
zlib	8	8	1
TypeScript	23	PositiveBlame	

Figure 16.1 Timings from running the Google Octane 2.0 Benchmark Suite. Column **Baseline** gives the baseline execution time of a run without contract assertion. Column **TreatJS** shows the execution time and the slowdown for running the benchmark program with contract monitoring.

Benchmark	Contract	Assert	Predicate	Membrane	Callback
Richards	24	1599377224	935751200	935751208	935751200
DeltaBlue	54	2320357672	1341331212	1341331220	1341331212
Crypto	1	5	3	11	3
RayTrace	42	687244882	509234422	509234430	509234422
EarleyBoyer	21	201	133	141	136
RegExp	0	0	0	8	0
Splay	10	11624671	7067593	7067601	7067593
SplayLatency	10	11624671	7067593	7067601	7067593
NavierStokes	51	48335	39109	39117	39109
pdf.js	824	1770208	1394694	1394702	1394694
Mandreel	4	14	5	13	8
MandreelLatency	4	14	5	13	8
Gameboy Emulator	3206	141669753	97487985	97487993	97488305
Code loading	5600	34800	18400	18408	18400
Box2DWeb	20075	180252500	112751947	112751955	112820587
zlib	0	0	0	8	0
TypeScript	730	7315	3902	3910	4068

Figure 16.2 Statistic from running the Google Octane 2.0 Benchmark Suite. Column **Contract** shows the number of top-level contract assertions. Column **Assert** contains the numbers of internal contract assertions whereby column **Predicate** lists the number of predicate evaluations. Column **Membrane** shows the numbers of wrap operations and the last column **Callback** gives the numbers of callback updates.

The examples show that the run-time impact of a contract depends on the frequency of its application. Contracts on a heavily used function (e.g., in *Richards*, *DeltaBlue*, or *RayTrace*) force a huge number of predicate checks, and thus they cause a significantly higher runtime deterioration than contracts on rarely used function.

16.2 Static Contract Simplification

Static Contract Simplification adapts ideas from previous work on hybrid contract checking [112] and static contract verification [83]. Its main objective is to use static contract checking to evaluate as much from a contract as possible and to collapse the remaining parts to a smaller contract that is more efficient to check at run-time.

To this end we propagate contracts through the program code, we detect and remove redundant parts, we check predicates where possible, and we propagate the remaining fragments to the enclosing module boundary. Finally, we condense the remaining fragments to new contracts which only contain parts that must be checked at run-time. Our goal is neither to detect contract violations at compile time nor to rewrite or optimize the source code of the program.

To demonstrate the essence of our idea, consider the following code snippet that contains a function with an intersection contract.

```
1 let addOne = ((λ plus (λ z ((plus 1) z)))
2               [(λ x (λ y (+ x y)))
3                @ ((Number? → (Number? → Number?)) ∩
4                   (String? → (String? → String?)))])
```

Listing 16.1 Definition of function `addOne` with an intersection contract.

The example creates a function `addOne` which returns the result of applying function `plus` to its argument `z` and the number value `1`. We use `@` to indicate a contract assertions, → stands for a function contract, and ∩ builds the intersection of two contracts. For readability, we put contract assertions in square brackets. Moreover, we rely on the previously defined flat contracts `Number?` and `String?`, which check for number and string values, respectively.

Assuming an overloaded `+` operator that works for number and string values (but fails for other inputs), a programmer may add an intersection contract to `plus`. The intersection contract enables the context (in this case function `addOne`) to use `plus` either with number or string values. It only requires that both values are of the same type. However, this requires to check the entire domain of both function contracts on every use of `plus`.

But, as the second argument of `plus` is already a number value we "statically" know that the right side of the intersection is never fulfilled. The only possibility to fulfill the contract is to call `plus` with a number value. Based on this knowledge it remains to check if the first argument of `plus` (`z`) and its return value satisfy the `Number?` contract. Moreover, as `z` is `addOne`'s parameter, we can lift the remaining contracts to an interface description on `addOne`, as the following example demonstrates. Only two predicate checks must remain, and this contract is cheaper to check as it has no intersection.

```
1 let addOne = [((λ plus (λ z ((plus 1) z))) (λ x (λ y (+ x y))))
2               @ (Number? → Number?)]
```

Listing 16.2 Definition of function `addOne` after static simplification.

This simplification can be done even without knowing if `addOne` is ever used. However, it still requires to track the evaluation state of statically evaluated predicates and to connect each

remaining fragment with its original contract operator. Moreover, it is necessary to preserve the responsibilities of a contract.

As the static contract simplification may change the order of predicate checks, we divided the simplification into two optimization levels: The *Baseline Simplification* and the *Subset Simplification*, each of which provides a different degree of simplification.

The Baseline Optimization unrolls and unfolds contracts and evaluates predicates where possible while preserving the blame behavior of a program. The *Subset Optimization* reorganizes contracts and forms new contracts from the remaining fragments, but thereby it may change the order of arising violations. Our contract simplification follows three overall guidelines, whereby each simplification step either satisfies strong or weak blame-preservation.

Real Optimization Each transformation step is a real benefit, i.e., it either *reduces* or *maintains* the total number of predicate checks at runtime. Simplification did never lead to more predicate checks.

Strong Blame-Preservation Each transformation step preserves the blame-behavior of a program, i.e., it maintains the order of contract checks at runtime. An optimized program results in exactly the same outcome as the original program.

Weak Blame-Preservation As some transformation steps may change the order of predicate checks they may also change the order of observed violations. It follows that an optimized program results in a blame state if and only if the original program would result in a blame state. However, while changing the order of predicate checks, an ill-behaved program might fail for another reason first.

Moreover, the simplification technique can also be used to over-approximate contract violations and to detect violations at compile-time. But this would violate all three guideline.

16.3 Implementation

We created an executable implementation using PLT Redex [37]. The implementation uses the contract calculus presented in Keil and Thiemann's work on Blame Assignment for Higher-Order Contracts [67] and allows to apply all simplification steps to a term with a contract. The implementation is available online at GitHub[1].

[1] `https://github.com/keil/Contract-Simplification`

17 Static Contract Simplification by Example

This chapter explains the main ideas of our simplification through a series of examples. We start with a simple example and work up to more complex ones.

17.1 Unrolling Delayed Contracts

The first series of examples considers different contracts on the `addOne` example from the previous chapter. To recap, the following code snippet defines `addOne` without contracts.

```
1 let addOne = ((λ plus (λ z ((plus 1) z))) (λ x (λ y (+ x y))))
```

Listing 17.1 Definition of function `addOne`.

The outermost term creates a function `addOne`, which returns the result of applying function `plus` to its argument `z` and the number value `1`. Function `plus` takes two arguments and applies the native `+` operator to it. In a first step, we add a simple function contract to `plus`.

```
1 let addOne = ((λ plus (λ z ((plus 1) z)))
2               [(λ x (λ y (+ x y))) @ (Number? → (Number? → Number?))])
```

Listing 17.2 Definition of function `addOne` with a simple contract.

Here, `@` stands for a contract assertion, → defines a function contract, and `Number?` is a flat contract that checks for number value.

While `Number?` is a flat contract that is checked immediately when asserted to a value, the function contract is delayed and must stay with the value until the value is used. Thus, calling `addOne` forces three predicate checks.

Now, instead of asserting a delayed contract, our *Baseline Simplification* unrolls a delayed contract to all uses of the contracted value. This step removes the delayed contract from `plus` and grafts it to all occurrences of `plus` in `addOne`.

```
1 let addOne = ((λ plus (λ z (([plus @ (Number? → (Number? → Number?))] 1) z)))
2               (λ x (λ y (+ x y))))
```

Listing 17.3 Definition of function `addOne` after contract unrolling.

This step performs no simplification, but it yields an equivalent program that is better suited for reducing the function contract. The next step unfolds the function contract on `plus` to its domain and range expression as the contract is on an expression that is in the application position. The following code snippet demonstrates the results after completely unfolding the contract, which is a combination of several unfolding steps.

```
1 let addOne = ((λ plus (λ z [((plus [1 @ Number?]) [z @ Number?]) @ Number?]))
2               (λ x (λ y (+ x y))))
```

Listing 17.4 Definition of function `addOne` after contract unfolding.

Now, all function contracts are completely decomposed into flat contracts which are spread over the domain and range of the function. Obviously, the contract system must remember the origin and the responsibility of the flat contract to blame the correct party.

After unfolding, there are several flat contracts applied to values. All such contracts can be checked statically and, if satisfied, removed from the program without changing the outcome of that program. For example, `1` satisfies `Number?`.

```
let addOne = ((λ plus (λ z [((plus 1) [z @ Number?]) @ Number?]))
              (λ x (λ y (+ x y))))
```

Listing 17.5 Definition of function `addOne` after evaluating flat contracts on values.

In the last step we push the remaining contract fragments outwards where possible. Like unrolling, this step performs no simplification, but it prepares for further simplification steps. The special ⊤ contract accepts any value.

```
let addOne = [((λ plus (λ z ((plus 1) [z @ Number?]))) @ (⊤ → (⊤ → Number?))
               (λ x (λ y (+ x y))))]
```

Listing 17.6 Definition of function `addOne` after pushing flat contracts.

Now, a new function contract is formulated in `addOne`. As this contract is left in an application, we can now unroll the contract to the argument values like before.

```
let addOne = [((λ plus (λ z ((plus 1) [z @ Number?])))
               (λ x (λ y (+ x y)))) @  (⊤ → Number?)]
```

Listing 17.7 Definition of function `addOne` after unrolling the lifted contract.

Afterward, the outermost function definition carries a function contract that checks the return of `addOne` and one contract is left on `addOne`'s argument `z`. However, as the Baseline Simplification guarantees *Strong Blame-Preservation*, it is not allowed to perform differently than the dynamics would do at run-time. No further simplifications are possible.

17.2 Treating Intersection and Union

Our next example considers the full *addOne* example from the previous chapter, which we repeat for convenience.

```
let addOne = ((λ plus (λ z ((plus 1) z)))
              [(λ x (λ y (+ x y)))
               @ ((Number? → (Number? → Number?)) ∩
                  (String? → (String? → String?)))])
```

Listing 17.8 Definition of function `addOne` with an intersection contract.

Here, `String?` is another flat contract that checks for string values. Like function contracts, the intersection of two function contracts is a *delayed* contract and must stay with the function until the function is used in an application. Moreover, for an intersection, the context can choose to fulfill the left or the right side of the intersection. Thus, every call to `addOne` triggers six predicate checks.

The *Baseline Simplification* unrolls the intersection contract to all uses of the contracted value and unfolds the function contracts to the domain and range when in an application position. The following code snippet demonstrates the unfolding of the intersection contract.

```
let addOne =
((λ plus
 (λ z [[((plus [[1 @₂ String?] @₁ Number?]) [[z @₂ String?] @₁ Number?])
        @₁ Number?] @₂ String?]))
 (λ x (λ y (+ x y))))
```

Listing 17.9 Definition of function `addOne` after unfolding the intersection contract.

Also, the static contract simplifier must maintain additional information in the background to connect each contract with the enclosing operation. The flat contracts `Number?` and `String?` still belong to different sides of the intersection and no value needs to fulfill both contracts at the same time. Here, $@_1$ and $@_2$ indicate the originating operand of the intersection. The order of the contracts corresponds to the dynamic evaluation at run-time.

Like before, the Baseline Simplification checks flat contracts applied to values. Here, `1` satisfies `Number?`, but violates `String?`. While `Number?` can now be removed, information about a failing contract must remain in the source program. In general, it is not possible to report a failure statically because we do not know if the code causing the violation (in this case, function `addOne`) is ever executed. However, we do not have to keep the original contract in the expression. For efficiency reasons, the transformation replaces it with a "false contract" $\bot$, whose sole duty is to remember a contract violation and report it at runtime. The following code snippet shows the result.

```
let addOne =
((λ plus
 (λ z [[((plus [1 @₂ ⊥]) [[z @₂ String?] @₁ Number?]) @₁ Number?] @₂ String?]))
 (λ x (λ y (+ x y))))
```

Listing 17.10 Definition of function `addOne` after evaluating flat contracts.

The next snippet shows the complete result after pushing the remaining fragments outwards.

```
let addOne =
[[((λ plus (λ z ((plus [1 @₂ ⊥]) [[z @₂ String?] @₁ Number?])))
   (λ x (λ y (+ x y)))) @₁ (⊤ → Number?)] @₂ (⊤ → String?)]
```

Listing 17.11 Definition of function `addOne` after pushing flat contracts.

Afterward, the outermost expression carries two function contracts that check the return value of `addOne`; two contracts are left in the body of `addOne` to test the argument.

17.3 Splitting Alternatives in Separated Observations

When remembering the last state in Section 17.2, we still check `String?` even though we know that right side of the intersection never succeeds. This is because the Baseline Simplification preserves the blame behavior of the underlying program and is thus not able to deal with alternatives. However, the *Subset Simplification* abandons the strong blame preservation to optimize contracts further. Instead of unfolding intersection and union contracts directly, we now observe both alternatives in separation and join the remaining contracts or discard a whole branch if it leads to a violation.

In our example, we see that `1` $@_2$ `String?` leads to a context failure of the right function contract of the intersection. Thus, it is useless to keep the other fragments of the right operand which yields a context violation. We only need to remember the first failing contract in this branch. The following code snippet shows the result.

```
let addOne =
[((λ plus (λ z ((plus [1 @₂ ⊥]) [z @₁ Number?])))
  (λ x (λ y (+ x y)))) @₁ (⊤ → Number?)]
```

Listing 17.12 Definition of function `addOne` after evaluating alternatives.

It remains to check `z` and `addOne`'s return to be a number. This are two remaining checks per use of `addOne`. However, we still need to keep the information on the failing alternative. This is required to throw an exception if also the other alternative fails.

17.4 Lifting Contracts

As shown by the previous examples, contracts on the function's body are reassembled to a new function contract, whereas contracts on function arguments must remain to preserve the order of predicate checks. Continuing the example, our next simplification step constructs new function contracts from the remaining contracts on the arguments.

```
let addOne =
[[(λ plus
  (λ z ((plus [1 @₂ ⊥]) z)))
  (λ x (λ y (+ x y)))) @₁ (Number? → ⊤)] @₁ (⊤ → Number?)]
```

Listing 17.13 Definition of function `addOne` after lifting contracts on arguments.

This *lifting* to the function expression will only reorganize existing contracts. The `addOne` function now contains another function contract that restricts its domain to number values. As this step puts contracts on arguments in front, it may change the order in which violations arise. However, it prepares for further simplifications.

17.5 Condensing Contracts

Continuing the example, the outermost function expression carries two function contracts which check the argument and the return value of `addOne`. Moreover, both contracts arise from the same source contract and belong to the same side of the intersection. Thus, these function contracts can be *condensed* to a single function contract on `addOne`.

```
let addOne =
[(λ plus
 (λ z ((plus [1 @₂ ⊥]) z)))
 (λ x (λ y (+ x y)))) @₁ (Number? → Number?)]
```

Listing 17.14 Definition of function `addOne` after condensing function contracts.

However, this step is only possible if the argument contract, here `Number?` $\to \top$, is in a negative position and if the return contract, $\top \to$ `Number?` is in a positive position. Otherwise, the new function contract would blame the wrong party for a contract violation.

17.6 Contract Subsets

Splitting intersection and union contracts into separate observations has another important advantage: we know that in every branch (observation) every contract must be fulfilled. We do not have any further alternatives. Based on this we can relate contracts to other contracts and reuse knowledge about already asserted contracts to reduce redundant checks. To demonstrate, we construct another version of our running example.

```
let addOne =
((λ plus [(λ z ((plus 1) z)) @ (Positive? → Positive?)])
 [(λ x (λ y (+ x y)))
  @ ((Number? → (Number? → Number?)) ∩
     (String? → (String? → String?)))])
```

Listing 17.15 Definition of function `addOne` with two function contracts.

Here, `Positive?` is another flat contract that checks for positive number values. In addition to the intersection contract on `plus`, `addOne` now contains another function contract which

requires `addOne` to be called with a positive number and to return a positive number. After performing the same transformation steps as before, we obtain the following expressions.

```
let addOne =
[[[((λ plus (λ z ((plus 1) z))) (λ x (λ y (+ x y))))
   @₁ (Number? → ⊤)] @₁ (⊤ → Number?)] @ (Positive? → Positive?)]
‖
let addOne =
[((λ plus (λ z ((plus [1 @₂ ⊥]) z))) (λ x (λ y (+ x y))))
 @ (Positive? → Positive?)]
```

Listing 17.16 Definition of function `addOne` in a split observation.

The intersection is split into two parallel observations, as indicated by ‖. All contracts were pushed to the outermost function expression and form a new specification for that function. However, some of the contract checks are redundant.

For example, the innermost contract on the first branch requires `addOne` to be called with a number value, whereas the outermost contract already restricts the argument to a positive number. As positive numbers are a proper subset of numbers, the inner contract will never raise a violation if the outer contract is satisfied. Moreover, the middle function contract requires `addOne` to return a number value, whereas the outermost contract requires a positive number value and is thus more restrictive.

To sum up, the contract `Positive?` → `Positive?` is more restrictive than both other contracts on `addOne`, that is, `Number?` → ⊤ and ⊤ → `Number?`. We say that a contract $\mathcal{C}$ is more restrictive than a contract $\mathcal{D}$, if and only if the satisfaction of $\mathcal{C}$ implies the satisfaction of $\mathcal{D}$.

However, if a more restrictive contract is violated, then this violation does not imply a violation of the less restrictive contract. But, as in each branch every contract must be fulfilled, the program already stops with a contract violation. There is no need to check a less restrictive contract on that branch.

Thus, both "less restrictive" contracts can be removed without changing the blame behavior of the program. In the first branch, the outermost function contract persists as it subsumes any other contract, whereas in the second branch both contracts remain.

```
let addOne =
[((λ plus (λ z ((plus 1) z))) (λ x (λ y (+ x y)))) @ (Positive? → Positive?)]
‖
let addOne =
[((λ plus (λ z ((plus [1 @₂ ⊥]) z)))
 (λ x (λ y (+ x y)))) @ (Positive? → Positive?)]
```

Listing 17.17 Definition of function `addOne` after removing redundant checks.

Unfortunately, removing the less restrictive contract might change the order of observed violations, as the less restrictive contract might report its violation first.

Finally, after finishing all transformation steps, the simplifications joins the split observation to a final expression.

```
let addOne =
[((λ plus
 (λ z ((plus [1 @₂ ⊥]) z)))
 (λ x (λ y (+ x y)))) @ (Positive? → Positive?)]
```

Listing 17.18 Definition of function `addOne` after joining split observations.

17.7 Propagating Blame

In the previous example, we have seen that contracts of a failing alternative can be removed and only one $\bot$ on the first failing term must remain. However, we are only allowed to remove contracts in the same function body in which the failing contract is. We cannot treat an entire contract as violated if the violation only occurs in a certain context of the program. But, $\bot$ is also a contract and we can still apply further simplification steps.

To demonstrate, we consider yet another variant of the `addOne` example.

```
1 let addOne = ((λ plus (λ z ((plus 1) z)))
2               [(λ x (λ y (+ x y))) @ (String? → (String? → String?))])
```

Listing 17.19 Definition of function `addOne` with a failing contract.

After applying some simplification steps we result in the following code snippet:

```
1 let addOne = ((λ plus (λ z ((plus [1 @ ⊥]) z))) (λ x (λ y (+ x y))))
```

Listing 17.20 Definition of function `addOne` after some simplification steps.

Now, all contracts that originate the function contract have been removed and only one $\bot$ remains in the function's body. However, as we know that each call of `addOne` will result in a contract violation, we can formalize this as a function contract on `addOne`.

```
1 let addOne = ((λ plus [(λ z ((plus 1) z)) @ (⊥ → ⊤)]) (λ x (λ y (+ x y))))
```

Listing 17.21 Definition of function `addOne` with a lifted function contract.

The new function contract reports the violation immediately when the function is used in an application. Like before, this only prepares to apply further simplification steps, i.e., we may propagate the function contract to the enclosing function definition, or we may unroll the function contract to all uses of the contracted function and therefore trigger a contract violation in another context. Like before, propagating is only allowed if we know that each call of the enclosing function also results in a call of the inner function. The following code snippet shows the result after propagating the "violation" to the outermost expression.

```
1 let addOne = [((λ plus (λ z ((plus 1) z))) (λ x (λ y (+ x y)))) @ (⊥ → ⊤)]
```

Listing 17.22 Definition of function `addOne` after propagating the function contract.

18 Practical Evaluation

To give an insight into the runtime improvements of our contract simplification, we apply the transformation to different versions of the `addOne` example from Chapter 17. Our testing procedure uses the `addOne` function to increase a counter on each iteration in a while loop.

18.1 The Example Programs

Lacking an implementation for JavaScript, we applied the simplification steps manually to different versions of the `addOne` example from Chapter 17. Each example program addresses a particular aspect and contains a different number of contracts. For example, *Example 7* corresponds the `addOne` function in Section 17.1, and *Example 9* corresponds to the function in Section 17.6. To run the examples, we use the TreatJS contract system for JavaScript and the SpiderMonkey[1] JavaScript engine.

The following listings show the JavaScript implementation of all benchmark programs.

▶ **Example 6.** In our first example, we add a simple function contract to function `plus` which restricts domain and range of `plus` to number values. Every use of `addOne1` forces three predicates checks.

```
let addOne1 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Function([typeNumber, typeNumber], typeNumber))
  let addOne = function (x) {
    return plus(x, 1);
  }
  return addOne;
})();
```

Listing 18.1 Definition of function `addOne1`.

▶ **Example 7.** Our second example adds an intersection contract to function `plus`. The contract enables to use `plus` either with number or string values. Every use of `addOne2` forces 6 predicate checks.

```
let addOne2 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Intersection(
      Contract.Function([typeNumber, typeNumber], typeNumber),
      Contract.Function([typeString, typeString], typeString)
  ));
  let addOne = function (x) {
    return plus(x, 1);
  }
  return addOne;
})();
```

Listing 18.2 Definition of function `addOne2`.

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

▶ **Example 8.** The third example extends `addOne1` by adding another function contract to `addOne`. The contract restricts `addOne`'s domain to natural numbers and requires a positive number as return. In combination, every use of `addOne3` forces 5 predicate checks.

```
let addOne3 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Function([typeNumber, typeNumber], typeNumber));
  let addOne =  Contract.assert(function (x) {
    return plus (x, 1);
  }, Contract.Function([Natural], Positive));
  return addOne;
})();
```

Listing 18.3 Definition of function `addOne3`.

▶ **Example 9.** The next example extends `addOne2` with another function contract on `addOne`. In combination, every use of `addOne4` forces 8 predicate checks.

```
let addOne4 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Intersection(
      Contract.Function([typeNumber, typeNumber], typeNumber),
      Contract.Function([typeString, typeString], typeString)
  ));
  let addOne = Contract.assert(function (x) {
    return plus (x, 1);
  }, Contract.Function([Natural], Positive));
  return addOne;
})();
```

Listing 18.4 Definition of function `addOne4`.

▶ **Example 10.** In our fifth example we also overload `addOne` by performing either string concatenation or addition, depending on `addOne`'s input. While adding an intersection contract to `addOne`, every use of `addOne5` leads to 10 predicate checks.

```
let addOne5 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Intersection(
      Contract.Function([typeNumber, typeNumber], typeNumber),
      Contract.Function([typeString, typeString], typeString)
  ));
  let addOne = Contract.assert(function (x) {
    return (typeof x == 'string') ? plus (x, '1') : plus (x, 1);
  }, Contract.Intersection(
       Contract.Function([Natural], Positive),
       Contract.Function([typeString], typeString)
  );
  return addOne;
})();
```

Listing 18.5 Definition of function `addOne5`.

Benchmark	**Normal** *time (ms)*	**Baseline** *time (ms)*	**Subset** *time (ms)*
Example 6	39404	27081 (-31.27%)	27043 (-31.37%)
Example 7	87143	58385 (-33.00%)	46085 (-47.12%)
Example 8	66474	54396 (-18.17%)	26518 (-60.11%)
Example 9	114468	85126 (-25.63%)	44633 (-61.01%)
Example 10	148249	107956 (-27.18%)	59970 (-59.55%)
Example 11	295682	200009 (-32.36%)	118579 (-59.90%)

Table 18.1 Runtime values from running the TreatJS contract system on the example programs. The table shows the execution time of a run with both JIT compilers enabled. Column **Normal** gives the baseline execution time of the unmodified program. Column **Baseline** and column **Subset** contain the execution time after applying the baseline or subset simplification, respectively. The value in parentheses indicates the improvement (in percent).

▶ **Example 11.** In our last example we add very fine-grained contracts to `plus` and `addOne`. In this case we state different properties in different function contracts and use intersections to combine those properties. Before simplifying the contract, every call of `addOne6` leads to a total number 17 predicate checks.

```
let addOne6 = (function () {
  let plus = Contract.assert(function (x, y) {
    return x + y;
  }, Contract.Intersection(
      Contract.Intersection(
        Contract.Function([typeNumber, typeNumber], typeNumber),
        Contract.Function([typeString, typeString], typeString)),
      Contract.Intersection(
        Contract.Intersection(
          Contract.Function([Natural, Positive], Positive),
          Contract.Function([Positive, Natural], Positive)),
        Contract.Function([Negative, Negative], Negative))));
  let addOne = Contract.assert(function (x) {
    return plus (x, 1);
  }, Contract.Function([_Natural], _Positive));
  return addOne;
})();
```

Listing 18.6 Definition of function `addOne5`.

18.2 Results

Table 18.1, Table 18.2, and Table 18.3 show the execution time required for a loop with `100000` iterations. The numbers indicate that the *Baseline Simplification* improves the run-time by approximately 28%, whereas the *Subset Simplification* makes an improvement up to 62%. The obtained run-time improvement depends directly on number of predicates during execution. Table 18.4 lists the total number of forced predicate checks.

Benchmark	Normal *time (ms)*	Baseline *time (ms)*	Subset *time (ms)*
Example 6	58906	40383 (-31.44%)	40271 (-31.64%)
Example 7	131184	87544 (-33.26%)	69492 (-47.03%)
Example 8	99776	81719 (-18.10%)	40218 (-59.69%)
Example 9	173037	128754 (-25.59%)	66359 (-61.65%)
Example 10	221619	160360 (-27.64%)	88369 (-60.13%)
Example 11	441176	278601 (-36.85%)	163272 (-62.99%)

Table 18.2 Runtime values from running the TreatJS contract system on the example programs. The table shows the execution time of a run without IonMonkey (but the Baseline JIT remains enabled). Column **Normal** gives the baseline execution time of the unmodified program. Column **Baseline** and column **Subset** contain the execution time after applying the baseline or subset simplification, respectively. The value in parentheses indicates the improvement (in percent).

Benchmark	Normal *time (ms)*	Baseline *time (ms)*	Subset *time (ms)*
Example 6	81125	56439 (-30.43%)	54945 (-32.27%)
Example 7	186069	124434 (-33.12%)	96271 (-48.26%)
Example 8	136728	111596 (-18.38%)	55186 (-59.64%)
Example 9	240724	179451 (-25.45%)	91034 (-62.18%)
Example 10	315184	225316 (-28.51%)	123852 (-60.71%)
Example 11	597406	404276 (-32.33%)	233124 (-60.98%)

Table 18.3 Runtime values from running the TreatJS contract system on the example programs. The table shows the execution time of a run without any JIT compilation. Column **Normal** gives the baseline execution time of the unmodified program. Column **Baseline** and column **Subset** contain the execution time after applying the baseline or subset simplification, respectively. The value in parentheses indicates the improvement (in percent).

Benchmark	Normal *predicates*	Baseline *predicates*	Subset *predicates*
Example 6	300000	200000 (-33.33%)	200000 (-33.33%)
Example 7	600000	400000 (-33.33%)	300000 (-50.00%)
Example 8	500000	400000 (-20.00%)	200000 (-60.00%)
Example 9	800000	600000 (-25.00%)	300000 (-62.50%)
Example 10	1000000	800000 (-20.00%)	500000 (-50.00%)
Example 11	1700000	1200000 (-29.41%)	700000 (-58.82%)

Table 18.4 Runtime values from running the TreatJS contract system on the example programs. The table shows the total number of predicate check during execution. Column **Normal** gives the number of the unmodified program. Column **Baseline** and column **Subset** contain the number of predicate checks after applying the baseline or subset simplification, respectively. The value in parentheses indicates the improvement (in percent).

V

Related Work and Conclusion

19 Related Work

This chapter highlights and discusses related work concerning contract systems, proxy objects, sandboxes, and contract optimization.

Higher-Order Contracts

Software contracts evolved from Floyd and Hoare's work on using preconditions, postconditions, and invariants to reason about the correctness and completeness of a computer program [46, 58]. Meyer's *Design by Contract*™ methodology [76] extends this concept and formalizes the specification of contracts for all elements of a software system and introduces the idea of monitoring these contracts while the program executes.

While first introduced with the design of the Eiffel programming language, Findler and Felleisen [41] brought contracts and contract monitoring into higher-order functional languages. Software contracts are particularly important for dynamically typed languages as these languages only provide memory safety and dynamic type safety. Out of the need to provide reliable software components, contracts have become a prominent mechanism and attracted a plethora of follow-up works that ranges from semantic investigations [9, 40] over studies on blame assignment [26, 108] to extensions in various directions: polymorphic contracts [3, 8], behavioral and temporal contracts [29, 32], etc.

Contract Validation

Contracts may be validated statically or dynamically. Static contract frameworks (e.g., ESC/Java [43]) rely on verification techniques such as model checking, theorem proving, or static program analyses to prove the adherence for a contract. Others, e.g. [113, 104], use symbolic execution and automated reasoning to verify software contracts.

Purely static contract checking is advantageous as it avoids additional runtime costs, but existing approaches are either incomplete or limited to a restricted set of properties. Moreover, static contract checking does not work for a dynamic scripting language like JavaScript. Given guarantees does not apply to code loaded at runtime or injected via `eval`.

Dynamic contract monitoring, as proposed in Meyer's work, enables programmers to specify contracts for all components of a program without restricting the flexibility of the underlying programming language. However, runtime monitoring degrades the execution time of a program as it extends the original program with contract code which needs to be checked while the program executes.

Monitoring Semantics

In general, the literature comes with three distinct blame semantics: *lax*, *picky*, and *Indy*. All three approaches differ in the case of how a delayed contract monitor on a subject value behaves during the evaluation of another assertion. For example, one question is whether the domain contract on an argument value should be present while evaluating the range contract of a dependent function contract.

The so-called *Lax* semantics of Findler and Felleisen [41] erases the contract monitor on a subject value, whereas the *Picky* [10] semantics monitors the contract during the evaluation of another assertion. However, none of the semantics can be seen as correct **and** complete. The

lax semantics ignores contract violations caused by other contracts, and the *picky* semantics might blame the wrong party for a violation that happens while evaluating another contract.

To overcome this, the *Indy* semantics of Dimoulas et al. [26] reorganizes the contract monitor on a subject value such that it treats other contracts as a third party with its own obligations. This concept enables that each monitor on a subject value remains active during the evaluation of another contract, but all observed violations during the evaluation of the contract will blame the contract itself instead of the context or the subject value.

TreatJS provides a generalization of all three monitoring semantics. It reorganizes all contract monitors on values that flow into the sandbox of another contract. In addition, it requires to keep the compatibility of contracts into account as it must not mix up contracts from different sides of an intersection or union.

Contract Operators

Over time, contract facilities have been extended in line with constructions studied in type theory. This analogy enables to transfer specifications and desired behavior from statically checked type systems to dynamically checked contract systems. So, there are contract operators analogous to (dependent) function types [41], product types, sum types [57], universal types [3], as well as polymorphic types [8, 49]. However, there are no operators similar to intersection and union types.

Other inspirations came for example from boolean algebras. But, right now no contract system provides full support for arbitrary boolean combinations of contracts.

Combinations of Contracts

Dimoulas and Felleisen [25] propose a contract composition which is closely related to a conjunction of contracts. But their operator is restricted to contracts of the same type. Before evaluating the conjunction, it lifts the operator recursively to flat contracts where it finally builds the conjunction of the predicate results. Unfortunately, the lifting of a disjunction is not possible because this would change the meaning of the contract.

Racket's contract system [44, Chapter 8.1] provides a restricted version of conjunctions and disjunctions of contracts. The operators `and/c` and `or/c` on contracts are designed to extend the obvious behavior on flat contracts to higher-order contracts. The `and/c` contract tests a number of contracts, whereas the `or/c` contract tests in a predefined order from left to right if one of its contracts succeeds. The disjunction must resolve in a first-order-way which contract to choose. Thus, the disjunction fails unless exactly one contract succeeds.

Also, Racket provides an `case->` operator [44, Chapter 8.2] that builds a case distinction of different function contract. However, an arity check at assertion time must select one of the contracts such that the arity of each function contract must be different.

In summary, Racket's contract operators are significantly different from intersection and union. Their operators are inspired by disjunction and conjunction, but they place several restrictions on using the operators in combination with higher-order contracts. Lacking support for arbitrary combinations of higher-order contracts is a limitation of most existing contract systems. A function that accepts two kinds of argument values would require that both contracts are applied separately, at each call site of that function. But this scenario has consequences for the space-efficiency [56] of the underlying program.

In contrast, TreatJS enables to combine contracts into a single intersection or union contract. Our proposal for intersection and union contracts is grounded in the type-theoretic constructs for intersection and union types. Both operators implement real alternatives

without further restrictions and enable arbitrary combinations of contracts. However, this requires not to signal a violation immediately when a contract fails. The contract monitor must proceed and connect the outcome of each contract with the enclosing operation. Unfortunately, this makes contract monitoring more expensive.

Embedded Contract Language

Interface specification Languages like the Java Modeling Language (JML) [71] specify the behavior of a software component in terms of annotation comments to the source files. As the language itself usually ignores comments, annotation comments require special tools to verify the specification or to compile them into runtime verifications.

In contrast, a language-embedded contract language exploits the host language itself to state contracts. Contracts are first-class values that can be stored and reused for several applications. This approach is advantageous because it neither requires developers to learn a separate contract language nor is the contract system tied to a particular implementation. Contract code can use the full expressive power of a programming language, and there is no need to provide special compilers or tools.

However, there are also disadvantages. The contract execution may get entangled with the application code. As most language-embedded systems use simple host-language functions as predicates, a predicate may have side effects and interfere with the execution of the application code. As the contract code should not change the outcome of a contract abiding host program, it requires to keep the contract code away from the host program.

Sandboxing JavaScript

Web browsers usually come with browser-side protection mechanism such as the same-origin policy [93] or the signed script policy [98]. Both approaches follow an all-or-nothing-principle and do either allow code fragments to access the application state if the fragment comes from the same origin or a verified source, or it runs the fragments in isolation.

The most closely related work to our sandbox mechanism is the design of access control wrappers for revocable reference and membranes [21, 81]. A revocable reference is a proxy for a target object that can be instructed to detach from the target so that the target is no longer reachable and safe from effects. The fundamental property of both approaches is memory safety. In memory-safe programming languages, a reference can be seen as a transferable right to access an underlying object. If a value is not reachable, then it is safe from effects. However, a proxy membrane itself does not implement a sandbox as, until detached from the target, it usually forwards all operations to the target object.

Agten et al. [2] implement another JavaScript sandbox using proxies and membranes. Unlike our work, they place wrappers around sensitive data (e.g., DOM elements) to enforce policies, and they require that scripts are SES-compliant [94], a JavaScript subset that prohibits features that are either unsafe or that grant uncontrolled access. To execute scripts they use a SES-library and a language-embedded JavaScript parser that transforms non-compliant scripts at runtime. Unfortunately, doing this restricts the semantics if JavaScript's as it prohibits some of its dynamic features.

Arnaud et al. [4] provide a sandbox mechanism that is similar to the mechanism of TreatJS. Both use proxy objects to apply access restriction and to guarantee a side-effect free contract assertion. In both approaches, the access restrictions apply only to values that cross the sandbox boundary. However, neither of them implements a full-blown sandbox, because writing is completely forbidden and always leads to an exception. DecentJS, in contrast,

guarantees read-only access to the target object and enables sandbox internal modifications through shadow objects.

Patil et al. [86] present *JCShadow*, a reference monitor implemented as a Firefox extension. Their tool provides fine-grained access control to JavaScript resources. Like DecentJS, they implement shadow scopes to isolate scripts from each other and to regulate object access. Unlike DecentJS, *JCShadow* achieves a better runtime performance. While more efficient, their approach is platform-dependent as it is tied to a specific engine and it requires active maintenance to keep the implementation in sync with the development of the engine. DecentJS, in contrast, is implemented as a JavaScript library based on JavaScript's reflection API, which is part of the standard.

Most other approaches (e.g., [47, 79, 35, 1]) implement restrictions by filtering and rewriting untrusted JavaScript code or by removing features that enable uncontrolled access. For example, Caja [47, 79] compiles JavaScript code in a sanitized JavaScript subset that can safely be executed on normal engines. However, to function meaningful, Caja restricts all dynamic features and rewrites the code to a "cajoled" version with additional runtime checks that prevent access to potentially unsafe function and objects.

Reachability

Static approaches come with some drawbacks, as shown by a number of scientific papers [74, 39, 89]. First, they either restrict the dynamic features of JavaScript, or their guarantees do not apply to the code generated at runtime. Second, maintenance requires a lot of effort because the implementation becomes obsolete as the language evolves.

Thus, dynamic effect monitoring and dynamic access control play an important role in the context of JavaScript security, as shown by many authors [4, 20, 81, 64]. Disjoint reachability has been the theoretic fundamental of all these approaches. The principle has widespread use and provides a wealth of features, e.g., it is used to grant access rights [80, 78, 42], to enforce policies [48, 27], or for parallelism [60].

Effect Monitoring

Richards et al. [92] provide a WebKit implementation to monitor JavaScript programs at runtime. Rather than performing syntactic checks, they look at effects for history-based access control and to revoke the effects that violate policies implemented in C++.

Transcript, a Firefox extension by Dhawan et al. [24], extends JavaScript with support for transactions and speculative DOM updates. Similar to DecentJS, it builds a transactional scope and permits the execution of unrestricted guest code. Effects within a transaction are logged for inspection by the host program. They also provide features to commit updates and to recover from effects of malicious guest code.

JSConTest [54] is a JavaScript framework that helps to investigate the effects of unfamiliar JavaScript code through access permission contracts. It monitors the execution of JavaScript code and summarizes the observed access traces to a concise effect description. However, as JSConTest is implemented as an offline source-code transformation it has known omissions: there is no support for `with` and `eval`, it does not apply to code loaded at runtime, and the transformation becomes obsolete when the language evolves.

JSConTest2 [64] is a redesign and a reimplementation of JSConTest which monitors read and write operations on objects using JavaScript proxies. Similar to delayed contract checking, each proxy object contains an access permission contract that specifies a set of permitted access paths starting from that object. JSConTest2 addresses some shortcomings

of the previous version: it works for the full JavaScript language including all dynamic features and runtime code generation using `eval`. Compared to our sandbox, JSConTest2 implements a proxy membrane to encapsulate sensitive data. However, its typical use is to restrict access to an object rather than encapsulating dubious JavaScript code. But, it might be used in combination with DecentJS to implement policies that further restrict access to objects visible to the sandbox.

Language-embedded Sandboxes

JSFlow [53] is a full language-embedded JavaScript interpreter that enforces information flow policies at runtime. Like DecentJS, *JSFlow* itself is implemented in JavaScript. However, compared to DecentJS, the *JSFlow* interpreter causes a significantly higher runtime impact than our sandbox, which only re-implements the JavaScript semantics on the membrane.

A similar slowdown is reported for js.js [103], another language-embedded JavaScript interpreter conceived to execute untrusted JavaScript code. Its implementation provides a wealth of powerful features similar to DecentJS: fine-grained access control, support for the full JavaScript language, and full browser compatibility. However, its average slowdown factor is in a range of 100 to 200, and thus it is significantly higher than DecentJS's.

JavaScript Proxies

The JavaScript proxy API [20] is a JavaScript extension released in the ECMAScript 6 [33] standard. The API enables a developer to wrap an object in a proxy and to fully interpose all operations on the object, including property lookup, property update, and function applications on function objects. Proxies are particularly important to implement dynamic monitoring facilities on objects and functions as they avoid much of the shortcomings of static analysis techniques and offline code transformations. Especially in JavaScript static approaches are often lacking because of JavaScript's dynamic features.

Thus, many JavaScript systems use proxies to enforce guarantees at runtime. Prominent examples are the implementation of revocable references [21] and access control wrappers [64, 2], the implementation of contract wrappers [30], and the implementation of cross-compartment wrappers in SpiderMonkey's compartment concept [109].

In other languages, proxies have already been used to encapsulate components and to enforce policies [95] as well as for other dynamic effect systems, meta-level extension, behavioral reflection, security, or concurrency control [80, 5, 12].

Proxy Object and Equality

One issue with JavaScript proxies is that proxies are always different from their wrapped target objects. This issue makes it impossible to use JavaScript proxies as contract wrappers, because the introduction of contracts may influence the meaning of a program.

Racket's contract system [44, Chapter 8] reports similar problems with noninterference as we do. But, in Racket, there are two equality operators: `eq?` and `equal?`. The first operator implements pointer equality on objects and is similar to JavaScript's strict equality operator `===`, whereas the second operator performs structural equality. Racket's proxies are not transparent with respect to the `eq?` operator, but they are transparent with respect to `equal?`. So, the effect of non-transparent proxies is limited to a small number of programs because proxies are transparent with respect to structural equality.

Unfortunately, JavaScript does not have a structural equality operator like `equal?`. The only safe way to change the transparency of a proxy object is to modify the underlying

JavaScript engine. Other proposals, for example, the use of the macro system SweetJS [31] to rewrite all equality tests to a proxy-aware operation, do not work in combination with `eval` and hence they do not provide complete interposition.

Furthermore, the PLT group proposes two kinds of proxies, *chaperones* and *impersonators* [100]. Both differ in the degree of freedom for redefining the semantics of the underlying target object. So, a behavior-equal transparent proxy would be a great extension to JavaScript.

Contract Simplification

Nguyen et al. [83] present a static contract checker that has evolved from Tobin-Hochstadt and Van Horn's work on symbolic execution [104]. Their approach is to verify contracts by executing programs on *unknown* abstract values that are refined by contracts. If a function meets its obligation, the corresponding contract gets removed. However, their approach only applies to the positive side of a function contract.

Our approach on static contract simplification is closely related to symbolic executions. Both approaches refine values by contracts, either by moving the contract through the term or by using an abstract value. However, compared to our work, Nguyen et al. address the opposite direction. Where we unroll a contract to its enclosing context and decompose a contract into its components, they verify the function's obligations based on the given domain contract. In case that the given contract cannot be verified, they retain the contract, and the whole contract must remain at its original position. They are not able to simplify the domain portion of a function contract. Moreover, their symbolic verification is not able to handle true alternatives in the style of intersection and union contracts.

We claim that both approaches are complementary to one another. Our contract simplification would benefit from a preceding verification that simplifies the function's obligations before unrolling a contract to the enclosing context.

Xu [112] presents another approach that combines static and dynamic contract checking. Her approach translates contracts into verification conditions to be verified statically. Whereas satisfied conditions will be removed, there may be conditions that cannot be proved: they remain in the source program in the form of dynamic checks.

20 Future Work

We have shown the design and implementation of TreatJS, a language embedded, higher-order contract system for JavaScript which enforces contracts by runtime monitoring. Along the way, we propose an alternative design for transparent JavaScript proxies that are better suited to implement contract wrappers, and we have shown how to run JavaScript code in a configurable degree of isolation to the host application using proxy membranes.

However, to bring user-friendly and efficient contracts to JavaScript, there are still some issues and open research topics. This chapter identifies several directions open for future work and extension of the TreatJS contract system.

20.1 Precise Blame Messages

The presence of intersection and union contracts complicates the computation of precise error messages. This is because the adherence of a contract depends on failures and successes in different sub-contracts. A failing subcontract does not automatically lead to contract violation of the top-level assertion. Therefore, it is not possible to report a single value for a failing predicate. We must always consider the entire contract including all predicates.

To demonstrate, consider the following example with an intersection contract.

```
let addOne = Contract.assert(function addOne(x) {
  return x+"1";
}, Contract.Intersection.from(
  Contract.Function([typeNumber], typeNumber),
  Contract.Function([typeString], typeString)
));
addOne(1); // throws :PositiveBlame: Subject @ (({0:#typeNumber} → #typeNumber) -
    ({0:#typeString} → #typeString))
```

Listing 20.1 Example of a blame message.

In this examples, the context decides to call `addOne` with a number value. As the context can choose to use `addOne` either with number or string values, the violation of `typeString` on the domain did not lead to a contract violation. However, the subject value must deal with both inputs, and as it returns a string for a given number value it violates the intersection.

To sum up, neither the input value `1` nor the return `11` can be labeled as right or wrong in general. The intersection has two satisfied, and two violated predicates. But only the violation of `typeNumber` on the range of the left function contract leads to a top-level contract violation. This is because `typeNumber` on the domain is satisfied, so the function must return a number.

Alternatives require that the contract monitor connects each contract with the enclosing operation. This connection creates a structure for computing positive and negative blame according to the semantics of subject and context satisfaction, respectively. However, the error message is still imprecise as there are several combinations to violate the contract.

So, a next step could be to develop an algorithm that computes precise error messages. An error message should be clear and concise, and developers should be able to understand what happens and how to recover the error immediately.

20.2 Built-in Contract Syntax

Another issue shown by the example in Listing 20.1 is that writing contracts is awkward and reduces the readability of the underlying program code.

So, having smoother contract definitions would improve the acceptance of contracts. Contract assertions should be brief and concise. Possible solutions are to use a preceding contract compiler or by using a macro system like *sweet.js*. However, the biggest benefit would come from a built-in contract syntax which is part of the JavaScript standard.

20.3 Native Contract Proxies

We already examined the issue with transparency in various use cases of JavaScript proxies, and we showed that a significant number of object comparisons would fail when gradually adding contracts to a program. Therefore, we propose an alternative design for transparent proxies that is better suited for implementing a contract system like TreatJS. However, the presented transparent proxy is a straightforward extension of the already existing opaque proxy, i.e., it provides the same features, and it enables to override the same handler traps as the opaque counterpart.

But this feature is also disadvantageous. Proxies may redefine the semantics of the underlying target object arbitrarily, and thus they prevent the optimizing compilers from optimizing a program efficiently. Examples from Section 7 show that the sole introduction of a simple forwarding proxy degrades the execution time of a JavaScript program dramatically.

However, Chapter 15 already shows that it is sufficient to restrict contract proxies to projections. A native observer proxy that implements a projection could be more efficient as it does not change the semantics of the underlying target object. JIT compilers would still be able to optimize a program as usual. To make dynamic contract monitoring efficient, special contract proxies are essential to improve the runtime costs of contract monitoring.

20.4 Realm-aware Pure Functions

TreatJS uses normal JavaScript functions as predicates. However, to guarantee noninterference with the actual program execution, it executes predicate code in a sandbox. But this sandboxing impacts the execution of the underlying program and it complicates the notion of contracts because each reference must be imported before recompiling the function.

Determining the effects of a JavaScript function is nearly impossible as even a simple property access might be the call of a side-effecting getter function or the call of a handler trap which causes an undesired behavior. Because of this flexibility, JavaScript would benefit from a new function constructor that implements a pure function. Such a constructor must be part of the standard and implemented in the runtime system.

A pure function is a function that only maps its input into an output value without causing any observable side effects. A pure function can inspect its input, and it can evaluate pure expressions, including function calls of pure functions and the access to a property that is bound to getter (if this getter is also a pure function). Moreover, JavaScript proxies would also benefit as handler traps can be restricted to pure functions to implement an observer proxy that does not change the semantics of the target object.

To grant effects on certain objects, pure functions could be realm-aware, i.e., they are bound to a certain realm in which effects are permitted. One example of such a realm could be the constraint set of a contract monitor, which might be influenced by the evaluation of a handler trap.

20.5 Further Contract Constructs

The usability of a contract system is strongly influenced by the variety and quantity of contract operators. Over time, contract facilities in programming languages have been extended in line with constructions studied in type theory. There are contract operators equivalent to dependent (function) types [41], intersection and union types [67], product types, sum type [57], universal types [3], and recursive types [104]. Other inspirations come for example from logical operations or temporal logics such as LTL. However, there are still some constructs missing a contract system would benefit from:

Boolean Operators Right now there is no contract system which provides full support for boolean combinations of contracts. TreatJS provides a draft implementation of boolean operators, but its implementation does not strongly correspond to the actions of standard boolean operators. Whereas the definition of conjunction and disjunction of flat contracts is obvious, the meaning of negation and the meaning of boolean combinations on higher-order contracts is too vague. So, for example, it is not clear who to blame if one operand of a disjunction blames the context whereas the other operand blames the subject. Moreover, TreatJS's definition of boolean combinations violates two ground rules of a boolean algebra: *De Morgan's laws* and *double negation*.

Temporal Contracts TreatJS already provides a draft implementation of temporal contracts in the style of Disney et al.'s [32] definition of temporal higher-order contracts. The implementation uses a regular effect grammar over effects to specified temporal properties. However, the definition only allows combining behavioral and temporal contract at the top level. A fine-grained combination is not possible.

Stateful contracts By default, all contracts are stateless, i.e., their evaluation only depends on the subject value itself and other imported values, e.g., the bound argument value of a dependent function contract. In contrast, a stateful contract encompasses an internal state which, for example, might be used to remember previous evaluations.

Stateful contracts are particularly important in security-related applications. For example, an object contract might specify that the context of an object is only allowed to access either property `a` or `b`, but never both. Such a contract must remember the first access and, depending on this, decide whether a second access is allowed or not.

TreatJS already enables to write stateful predicates using contract constructors. However, their use and meaning are not always clear as a contract might fail for an incomprehensible reason, and it is not clear how to satisfy the contract.

20.6 Static Contract Simplification

Run-time monitoring of contracts impacts the execution time of the underlying program by the insertion of contract checks as well as by the introduction of proxy objects that perform delayed contract checks on demand. However, experiments from Chapter 16 show that it is not necessary to check the entire contract at runtime. Some contracts can be verified at compile-time and others can be simplified to smaller contracts that are collectively cheaper to check at runtime.

For example, a hybrid contract monitor that combines techniques from Chapter 17 and Nguyen et al.'s work [83] on soft contract verification could statically pre-evaluate a contract and reduce the unevaluated parts to a cheaper contract that only contains parts which must be checked at runtime. Such a system would make contracts more efficient while reducing the runtime costs and still preserving the flexibility and expressiveness of the contract system.

21 Conclusion

This work presents the design and implementation of TreatJS, a language embedded, higher-order contract system for JavaScript [33] which enforces contracts by runtime monitoring. Beyond the standard abstractions for higher-order contracts (flat contract, function contracts, dependent contracts), TreatJS provides contract abstraction and arbitrary combinations of contracts using intersection and union in combination with the following points:

- TreatJS provides a contract constructor that constructs and composes contracts at runtime using contract abstraction. A contract constructor may contain arbitrary JavaScript code, and it may encapsulate a local state. Contract constructors are the building blocks for dependent contracts, parameterized contracts, and recursive contracts.
- TreatJS's blame assignment for higher-order contracts with intersection and union has a number of novel aspects. First, it uses constraints to create a structure for computing positive and negative blame according to the semantics of subject and context satisfaction, respectively. Second, it applies a compatibility check to distinguish contracts from different sides of an intersection or union, and third, it provides three general monitoring semantics that handle the visibility of contracts inside of predicate code.
- TreatJS gives noninterference a high priority and proposes an implementation that enforces it. It employs a membrane-based sandbox to keep the predicate code apart from the normal program execution, and it encapsulates objects that are passed through the membrane to enforce write protection and to withhold external bindings from functions. Contracts are guaranteed to exert no side effects on a contract abiding program execution.
- TreatJS is implemented as a library in JavaScript. It enables a developer to specify all aspects of a contract using the full JavaScript language. Proxies implement delayed contract checking of function and object contracts and guarantee full interposition for the full JavaScript language, including the `with`-statement and `eval`.
- The implementation of TreatJS illustrates the need for another proxy constructor that is better suited for the implementation of contract wrappers. One issue with the actual contract implementation arises if a contract wrapper is different (not pointer-equal) from the target object so that an equality test between wrapper and target returns false instead of true. Thus, TreatJS comes with an implementation of a transparent object proxy which ensures transparent operations with all JavaScript programs.

We further presented the implementation of DecentJS, a language-embedded sandbox for full JavaScript. DecentJS uses JavaScript proxies and membranes to run JavaScript code in isolation to the application state. Proxies implement effect logging on the sandbox membrane, and they implement shadow objects to enable sandbox internal modifications of that object. The sandbox itself is written in JavaScript and thus provided as a library. All aspects are accessible through a sandbox API.

We also measured the impact of contract monitoring on the execution time of JavaScript programs. The experiments show that the runtime impact of a contract monitor depends on the particular value that is monitored and on the frequency with which the contracted values are used. While some programs' runtime is heavily impacted, others are nearly unaffected.

The case of this impact is **1.** that every contract extends the original source with additional contract code that needs to be checked when the program executes, **2.** that the contract monitor itself causes some runtime overhead and **3.** that the introduction of proxy objects that perform delayed contract checking prevents the optimizing compilers to optimize program

code involving contracts. Moreover, contracts may end up on hot paths in a program. We believe that carefully added contracts with the purpose of determining interface specifications will not seriously influence the run time of the underlying program.

As another novel aspect, we presented the idea of static contract simplification, which simplifies contracts that cannot be verified entirely at compile time. Contract fragments that cannot be verified stay in the source program and get lifted to the enclosing interface.

To this end, we specify two sets of transformation rules that give a focus on weak and strong blame preservation. They use a subcontract relation to reduce contracts that are subsumed by other contracts, and they split alternatives into separated observation.

Case studies with microbenchmarks show that a hybrid monitor that is based on static simplification techniques can decrease the total number of predicate checks at runtime and thus it improves the runtime impact caused by contracts and contract monitoring. Moreover, the case studies also show that the improvement depends on the granularity of contracts. More fine-grained contracts enable better improvements.

References

1 Adsafe: Making JavaScript safe for advertising. `http://www.adsafe.org/`, 2015.

2 Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 1–10. ACM, 2012.

3 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In Ball and Sagiv [6], pages 201–214.

4 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. Springer, 2010.

5 Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 921–938. ACM, 2011.

6 Thomas Ball and Mooly Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.

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

8 João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2011.

9 Matthias Blume and David A. McAllester. A sound (and complete) model of contracts. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 189–200. ACM, 2004.

10 Matthias Blume and David A. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

11 John Tang Boyland, editor. *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

12 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, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 331–344. ACM, 2004.

13 Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.

14 Chakra JavaScript engine. `https://github.com/Microsoft/ChakraCore`.

15 Arthur Charguéraud. Pretty-big-step semantics. In Felleisen and Gardner [38], pages 41–60.

16 Olaf Chitil. Practical typed lazy contracts. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 67–76. ACM, 2012.

17 Contract.js. `https://github.com/disnet/contracts.js/`.

18 contracts.ruby. `https://github.com/egonSchiele/contracts.ruby/`.

19 Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978.

20 Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In William D. Clinger, editor, *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, pages 59–72. ACM, 2010.

21 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178. Springer, 2013.

22 Dart. `https://www.dartlang.org`.

23 Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: sandboxing JavaScript to fight malicious websites. In Shin et al. [97], pages 1859–1864.

24 Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 382–391. IEEE Computer Society, 2009.

25 Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16:1–16:29, 2011.

26 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In Ball and Sagiv [6], pages 215–226.

27 Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 3–17. IEEE Computer Society, 2014.

28 Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. Future contracts. In António Porto and Francisco Javier López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 195–206. ACM, 2009.

29 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 214–233. Springer, 2012.

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

31 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'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 35–44. ACM, 2014.

32 Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 176–188. ACM, 2011.

33 ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, April 2015. ECMA International, ECMA-262, 6th edition.

34 Facebook. http://www.facebook.com/, 2015.

35 Facebook SDK for JavaScript. https://developers.facebook.com/docs/javascript/, 2015.

36 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Shin et al. [97], pages 2103–2110.

37 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

38 Matthias Felleisen and Philippa Gardner, editors. *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*. Springer, 2013.

39 Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: isolating proxied content. In Lex Stein and Alan Mislove, editors, *Proceedings of the 1st Workshop on Social Network Systems, SNS 2008, Glasgow, Scotland, UK, April 1, 2008*, pages 25–30. ACM, 2008.

40 Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In Hagiya and Wadler [51], pages 226–241.

41 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming ICFP '02, Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 48–59. ACM, 2002.

42 Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in Java. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 161–174. ACM, 2008.

43 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002.

44 Matthew Flatt. The Racket Reference. http://docs.racket-lang.org/reference/index.html, February 2017. Version 6.8.

45 Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. http://docs.racket-lang.org/guide/index.html, February 2017. Version 6.8.

46 Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

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

48 Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

49 Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In Pascal Costanza and Robert Hirschfeld, editors, *Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal, Quebec, Canada*, pages 29–40. ACM, 2007.

50 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010.

51 Masami Hagiya and Philip Wadler, editors. *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*. Springer, 2006.

52 David R. Hanson and Todd A. Proebsting. Dynamic variables. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 264–273. ACM, 2001.

53 Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1663–1671. ACM, 2014.

54 Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 111–122. ACM, 2012.

55 Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In Akinori Yonezawa, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25, 1990, Proceedings.*, pages 169–180. ACM, 1990.

56 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In Marco T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007.*, volume 8 of *Trends in Functional Programming*, pages 1–18. Intellect, 2007.

57 Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In Hagiya and Wadler [51], pages 208–225.

58 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

59 JavaScriptCore JavaScript engine. `https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore`.

60 James Christopher Jenista, Yong Hun Eom, and Brian Demsky. Using disjoint reachability for parallelization. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 198–224. Springer, 2011.

61 jscontract. `http://kinsey.no/projects/jsContract/`.

62 Murat Karaorman, Urs Hölzle, and John L. Bruno. jContractor: A reflective Java library to support design by contract. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99, Saint-Malo, France, July 19-21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 1999.

63 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In Boyland [11], pages 149–173.

64 Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. In Antony L. Hosking, Patrick Th. Eugster, and Carl Friedrich Bolz, editors,

DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 49–60. ACM, 2013.

65 Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. *CoRR*, abs/1312.3184, 2013.

66 Matthias Keil and Peter Thiemann. Type-based dependency analysis for JavaScript. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 47–58. ACM, 2013.

67 Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 375–386. ACM, 2015.

68 Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScripts. In Boyland [11], pages 28–51.

69 Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

70 R. Kramer. iContract - the Java(tm) design by contract(tm) tool. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, pages 295–307. IEEE Computer Society, 1998.

71 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*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*, pages 175–188. Springer, 1999.

72 Gary T. Leavens and Matthew B. Dwyer, editors. *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. ACM, 2012.

73 Luacontractor. `http://luaforge.net/projects/luacontractor/`.

74 Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91. IEEE Computer Society, 2009.

75 Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting JavaScript. In Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, editors, *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, volume 7127 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2010.

76 Bertrand Meyer. *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988.

77 Leo A. Meyerovich and V. Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 481–496. IEEE Computer Society, 2010.

78 Mark S. Miller, James E. Donnelley, and Alan H. Karp. Delegating responsibility in digital systems: Horton's "who done it?". In *2nd USENIX Workshop on Hot Topics in Security, HotSec'07, Boston, MA, USA, August 7, 2007*. USENIX Association, 2007.

79 Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript, 2008. Google White Paper.

80 Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In Felleisen and Gardner [38], pages 1–20.

81 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.

82 Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in AmbientTalk. *Softw., Pract. Exper.*, 39(7):661–699, 2009.

83 Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 139–152. ACM, 2014.

84 Node.js. `https://nodejs.org`.

85 Octane 2.0. `https://developers.google.com/octane/`.

86 Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards fine-grained access control in JavaScript contexts. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 720–729. IEEE Computer Society, 2011.

87 php-by-contract. `https://github.com/wick-ed/php-by-contract/`.

88 Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pages 47–60. ACM, 2009.

89 Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

90 Programming by contract for Python. `http://legacy.python.org/dev/peps/pep-0316/`.

91 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.

92 Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible access control for JavaScript. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 305–322. ACM, 2013.

93 Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy/`, 2008.

94 SecureEcmaScript (ses). `https://code.google.com/p/es-lab/wiki/SecureEcmaScript`, 2015.

95 Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box pi, wrappers, and causality. *Journal of Computer Security*, 11(2):135–188, 2003.

96 Nir Shavit and Dan Touitou. Software transactional memory. In James H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995.

97 Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors. *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*. ACM, 2010.

98 Signed scripts in Mozilla. `http://www-archive.mozilla.org/projects/security/components/signed-scripts.html/`, 2015.

99 SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

100 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In Leavens and Dwyer [72], pages 943–962.

101 Sweet.js. `http://sweetjs.org/`.

102 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 456–468. ACM, 2016.

103 Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. Javascript in JavaScript (js.js): Sandboxing third-party scripts. In E. Michael Maximilien, editor, *3rd USENIX Conference on Web Application Development, WebApps'12, Boston, MA, USA, June 13, 2012*, pages 95–100. USENIX Association, 2012.

104 Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Leavens and Dwyer [72], pages 537–554.

105 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.

106 TypeScript. `https://www.typescriptlang.org`.

107 V8 JavaScript engine. `https://developers.google.com/v8/`.

108 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.

109 Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Compartmental memory management in a modern web browser. In Hans-Juergen Boehm and David F. Bacon, editors, *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 119–128. ACM, 2011.

110 Alessandro Warth, Milan Stanojevic, and Todd D. Millstein. Statically scoped object adaptation with expanders. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 37–56. ACM, 2006.

111 Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

112 Dana N. Xu. Hybrid contract checking via symbolic simplification. In Oleg Kiselyov and Simon J. Thompson, editors, *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 107–116. ACM, 2012.

113 Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for haskell. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 41–52. ACM, 2009.

Index

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, date

Roman Matthias Keil