Research Statement

My research interests include static and dynamic program analysis, programming language design in general, theoretical computer science, and foundations of software technology.

Within this area, I focus on analyzing higher-order functional programs by enforcing program behavior and managing access control. In particular, my research interests aim at analysis techniques for JavaScript, which make substantial use of contract monitoring, effect monitoring, and dynamic effect inference to provide static and dynamic program guarantees.

Much of my previous work concerns the development of *TreatJS* [7], a language-embedded higher-order contract system for JavaScript. *TreatJS* provides many novel aspects of both an applied and theoretical nature. Moreover, I developed a language-embedded sandbox for JavaScript, which allows running JavaScript code in isolation to the host application, and I contributed to the theory of regular expressions, formal languages, and automaton theory.

Higher-Order Contracts For JavaScript

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 an intermediate language for other languages to target, such as TypeScript or Dart. Hence, it is no surprise that JavaScript is the focus of many research works, out of the need to create better development tools for JavaScript programmers and launch new language features.

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.
- Everything can be modified, from the fields and methods of an object over its prototype property to the scope chain of a function closure.

Consequently, JavaScript code is prone to injection attacks, library code can read and manipulate everything reachable from the global scope, and third-party code can access sensitive data. Furthermore, side effects may cause unexpected behavior, so program understanding and maintenance become difficult.

One possible solution to overcome these limitations is using contracts with run-time monitoring. Software contracts were introduced with Meyer's *Design by Contract*$^{\text{TM}}$ methodology, which stipulates invariants for objects and Hoare-like pre- and postconditions for functions.

Since Meyer's work, the contract idea has taken off and attracted a plethora of follow-up works that range from contract monitoring of higher-order functional languages over semantic investigations and studies on blame assignment to extensions in various directions: polymorphic contracts, behavioral and temporal contracts, etc.

Contract monitoring has become a prominent mechanism to provide solid guarantees for programs in dynamically typed languages while preserving their flexibility and expressiveness. Hence, the first higher-order contract systems were devised for Scheme and Racket, but other dynamic languages like JavaScript, Python, PHP, Ruby, and Lua have followed suit.

My research focuses on the design and implementation of *TreatJS* [7], a language-embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond the standard abstractions for higher-order contracts (flat contract, function contracts, dependent contracts), *TreatJS* comes with the following unique features:

– *TreatJS* provides a contract constructor that uses contract abstraction to construct and compose contracts at run-time. A contract constructor may contain arbitrary JavaScript code and encapsulate a local state. Contract constructors are the building blocks for a dependent, parameterized, and recursive contracts.

– *TreatJS* provides contract intersection and union operators similar to their type-theoretic counterparts. These operators enable developers to specify independent properties in independent contracts and combine them using intersection and union.

– *TreatJS*'s blame assignment for higher-order contracts with intersection and union [6] has several novel aspects. First, it uses constraints to create a structure for computing positive and negative blame according to subject and context satisfaction semantics, 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 handles the visibility of contracts inside predicate code.

– *TreatJS* gives noninterference a high priority. Its implementation 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 withhold external bindings from functions. Contracts are guaranteed not to exert side effects on 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 whole JavaScript language, including the with-statement and eval.

However, to bring user-friendly contracts to the JavaScript programming language, some issues and directions are still open for future work. The following sections address some open research topics I plan to continue in the short and medium-term.

Static Contract Simplification

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 [9, 7]. All existing contract systems like Racket's contract framework [3, Chapter 7], Disney's JavaScript contract system *contracts.js* [2], *JSConTest2* [5], or *TreatJS* [7] for JavaScript report a considerable slowdown when extending programs with contracts.

These costs arise because every contract extends a program with additional code that checks the contract while executing. Moreover, developers may add contracts at frequently used functions and objects on hot-paths in a program. In particular, predicates may repeatedly check the same values, and different predicates may check redundant parts.

In contrast, static contract checking [11] avoids 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 vast number of false positives.

My ongoing work on *Static Contract Simplification* attacks this issue with compile-time program transformation. It adapts ideas from previous work on hybrid contract checking [10] and static contract verification [8] to evaluate as much of a contract as possible and collapse the remaining parts to a smaller contract that is more efficient to check at run-time. To this end, we unroll contracts through the program code, detect and remove redundant parts, check predicates where possible, and lift the remaining fragments to the enclosing module boundary. Finally, we combine the remaining fragments to new contracts, which only contain parts that must be checked at run-time. Such a simplification can be done even without knowing the concrete execution of a program.

Native Contract Proxies

The implementation of *TreatJS* illustrates the need for a different proxy constructor that is better suited for implementing contract wrappers. One issue with the current contract implementation arises because 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* already implements a transparent object proxy [4], ensuring transparent operations with all JavaScript programs.

My work on Transparent Object Proxies for JavaScript [4] 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 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 enables the user to override the same traps as the opaque counterpart.

However, this power comes with some danger. Proxies may redefine the semantics of the underlying target object arbitrarily, and thus they prevent specific optimizations in a compiler. Examples from our work show that the sole introduction of simple forwarding proxies degrades the execution time of a JavaScript program dramatically.

In my previous research, I have already shown that it is sufficient to restrict contract proxies to projections [4]. 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. Therefore, to make dynamic contract monitoring efficient, special contract proxies are essential to improve the run-time costs of contract monitoring.

Realm-aware Pure Functions

TreatJS uses normal JavaScript functions as predicates and executes the predicate code in a sandbox to guarantee noninterference with the actual program execution. However, this sandboxing impacts the execution of the underlying program, and it complicates the writing of contracts because each needed reference must be imported into the sandbox.

In JavaScript, determining the effects of a function is nearly impossible as even simple property access might be the call of a side-effecting getter function or the call of a handler trap that causes an undesired behavior. Because of this flexibility, JavaScript would benefit from a new function constructor that implements a *pure function*.

A *pure function* is a function that only maps its input into an output 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 a getter function (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. Besides, pure functions could be realm-aware to grant effects on particular objects, i.e., they are bound to a specific realm in which effects are permitted. One example for such a realm could be the constraint set of a contract monitor, which might influence the evaluation of a handler trap.

Precise Blame Messages

The presence of intersection and union contracts complicates the computation of precise error messages because the compliance of such a contract depends on failures and successes in different sub-contracts. Moreover, a failing sub-contract does not automatically lead to contract violation of the top-level assertion. Therefore, it is impossible to report a single value that violates the contract: it always requires considering the entire contract, including all predicates.

The adherence of intersection and union requires that the contract monitor connects each contract with the enclosing operation. This connection creates a structure for computing positive and negative blame according to subject and context satisfaction semantics. Therefore, the next step could be to develop an algorithm that computes precise error messages based on this structure. An error message should be clear and concise, and developers should immediately understand what happens and how to recover the error.

Looking Forward

Static program analysis is the automated source code analysis performed without executing the program's source code. Due to its static nature, it compares more to testing the program's internal structure rather than functional testing. Therefore, static program analysis can be stated to be complete but lacks correctness as it might need to rely on source code abstraction.

The *dynamic program analysis* adopts the opposite approach and performs while the program executes. It involves analyzing the behavior of a program, and while its outcome always reflects the concrete execution correctly, it will never be complete. However, contrary to the static opinion, which is fixed in size and can be done at compile-time before executing the program's code, dynamic program analysis comes with the cost to slow down the execution of the underlying program. On the other side, one of its main benefits is that runtime analysis can detect vulnerabilities too subtle or complex for static analysis. This benefit makes dynamic program analysis particularly attractive for dynamic programming languages with less static guarantees like JavaScript. But also other programming languages like Java might benefit from extended dynamic program analysis to state more fine-grained invariants and refined program behavior.

In a nutshell, dynamic program analysis is essential, despite its increased runtime impact. Therefore, the question is not if we need dynamic program analysis in general; the question is about reducing the overhead of runtime monitoring by using dynamic program analysis most efficiently and how to combine static and dynamic program analysis to a hybrid solution combining the best from both worlds.

I plan to continue my work on static contract simplification in the short term. So far, I have implemented a prototype using PLT Redex, but there is as yet no implementation that works with JavaScript contracts. To this end, I plan to develop a compile-time program transformation that simplifies contract definitions to residual contracts that are collectively cheaper to check at runtime. This transformation (which is somehow related to hybrid contract checking) combines static and dynamic contract checking techniques and can also simplify contracts that cannot be verified entirely at compile time.

n a larger perspective, I plan to continue my work on contract systems and pursue my overall goal of bringing efficient and user-friendly contracts to JavaScript. Two concrete projects in this line are1. the development of more efficient contract wrappers that implement a projection, and 2. the design and implementation of a pure function construct for JavaScript.

Reference

[1] 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.

[2] Tim Disney. contracts.js. `https://github.com/disnet/contracts.js`, April 2013.

[3] Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. `http://docs.racket-lang.org/guide/index.html`, February 2017. Version 6.8.

[4] Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in javascript. In Boyland [1], pages 149–173.

[5] Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using javascript proxies. In 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.

[6] 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.

[7] Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In Boyland [1], pages 28–51.

[8] 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.

[9] 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.

[10] 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.

[11] 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.