JavaScript with Blame

Jakub Zalewski

Master of Science
School of Informatics
University of Edinburgh
2014
Abstract

Type information can be used to enhance the capabilities of dynamically typed languages, i.e. facilitate static analysis, serve as documentation and can increase runtime performance. This work aims to verify type declarations for TypeScript, a Gradually typed variant of JavaScript, by providing a command tool utility that allows wrap untyped code with type checking wrapper. It extends Blame Calculus — a formal framework for verifying types when mixing typed and untyped code — to cover arrays, objects and union types, and attempts to show that it can be translated to a small portion of JavaScript formalised basing on $\lambda_{JS}$. This formal model is then used to implement a comandline tool used to perform type checking on JavaScript modules. Notably, it is shown that the problem of unions for contracts can be solved.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Jakub Zalewski)
Contents

1 Introduction ................................................. 1
  1.1 Introduction and Purpose .............................. 1

2 Background .................................................. 2
  2.1 Contracts ................................................ 2
    2.1.1 First-Order Contracts ............................ 3
    2.1.2 Higher-Order Contracts ......................... 3
    2.1.3 Blame .............................................. 4
  2.2 Blame Calculus ......................................... 5
  2.3 JavaScript .............................................. 7
    2.3.1 TypeScript ......................................... 7
    2.3.2 JavaScript engines ............................... 7
  2.4 Proxies .................................................. 8
    2.4.1 Meta-programming ................................. 8
    2.4.2 Direct Proxies ................................... 9

3 Theory ....................................................... 10
  3.1 Simplified Blame Calculus ............................. 10
    3.1.1 Blame Paths ...................................... 12
    3.1.2 Objects .......................................... 12
    3.1.3 Arrays ........................................... 13
  3.2 The $\lambda_{JSP}$ Calculus ............................. 13
    3.2.1 Desugaring ...................................... 14

4 Implementation ............................................. 17
  4.1 Wrapper ............................................... 17
    4.1.1 Proxy-based Wrappers ........................... 17
    4.1.2 Describing Types ............................... 18
4.2 Wrap API ......................................................... 19
4.3 Wrapping Functions ........................................... 19
4.4 Wrapping Foralls ............................................... 20
  4.4.1 Dynamic Sealing ......................................... 20
4.5 Wrapping Arrays and Objects ............................... 21
4.6 Wrapping Unions ............................................... 21
4.7 Parser ............................................................ 22
  4.7.1 Extracting Type Information ............................ 22
4.8 Wrapper Generation ........................................... 23
  4.8.1 Dealing with Recursive Types ........................... 23
  4.8.2 Wrapping Modules ....................................... 23

5 Evaluation ....................................................... 24
  5.1 Acquiring Test Cases ........................................ 24
  5.2 Wrapping Modules .......................................... 24
  5.3 Running Unit Tests ......................................... 24
  5.4 Inspecting Results ......................................... 25
  5.5 Measuring modules Lines of Code ........................ 28

6 Conclusion ....................................................... 29

A Working model for union contracts in Haskell ........... 30
Chapter 1

Introduction

1.1 Introduction and Purpose

This project develops an interface wrapper between typed TypeScript code and untyped JavaScript code in a form of dynamic type checking tool based on Blame calculus invented by Wadler and Findler (2009) and later extended to incorporate parametric polymorphism by Ahmed et al. (2011).

Dynamically typed programming languages permit ill-typed programs and might get stuck on type errors, which is usually reported by the language runtime halting abruptly. This can be addressed by a sound static type checker, but type casts introduce ambiguity, which often cannot be resolved by static type checking.

There are attempts to integrate the advantages of dynamically typed programming languages into statically typed languages and vice versa.

TypeScript is a gradually typed programming language which compiles to JavaScript. It allows to interact with external untyped libraries by using type definitions provided by the developer, however those type definitions are accepted as-is, without verifying or enforcing the invariants specified in the type declarations.

The core contribution of the project is to extend Blame Calculus with the support of objects, arrays and unions.

The problem of unions is particularly interesting as the idea to solve it is novel and can be attributed to the author. Notably, it solves the problem of dealing with unions of higher-order contract (Tobin-Hochstadt and Van Horn, 2012).
Chapter 2

Background

Type information provides three main advantages to a program written in a dynamically typed programming language: (i) they allow to analyse the program statically, (ii) they provide documentation for the developer, and (iii) they increase the program’s performance.

Sound type system ensures that the program is well-typed, which guarantees that the program will not get stuck on a type error.

Type declarations also retain some of the original author’s understanding of the code for the developer who examines the provided files by eye, as noted by Reynolds (1985). Nowadays, developers are assisted by an Integrated Development Environment (IDE) — which scans the files for type declarations automatically — and displays a drop-down list of suggestions for the statement at hand, or displays a hint about the current declaration based on the type.

Finally, certain dynamically typed languages use type information to generate efficient native code and greatly increase the performance; notable examples include PyPy for Python (The PyPy Project, 2014) and asm.js for JavaScript (Herman et al., 2014).

2.1 Contracts

Parnas (1972) introduced the concept of verifying and theorising about modules in terms of specifications they expose and Meyer (1992) popularized “contracts” for imperative languages as a part of the “Design by Contract” philosophy for the Eiffel programming language.

First-order contracts efficiently ensure soundness of procedural programs, as
shown by a study conducted by Rosenblum (1995); they also ensure soundness of object-oriented programs, even in the presence of method overriding, as shown by Findler and Felleisen (2001). Higher-order contracts and blame were introduced by Findler and Felleisen (2002), and the remainder of this section explains their contribution.

2.1.1 First-Order Contracts

Contracts for first-order values evaluate a set of assertions to ensure that the supplied value conforms to a desired invariant. Consider the following example of a contract — written in representation established by Findler and Felleisen (2002) — that checks if a value is greater than 10, applied to a variable $i$.

\[ > 10 \] $i$

If the value of $i$ is greater than 10, the evaluation will continue uninterrupted, and if the value of $i$ is less or equals than 10, the evaluation will be interrupted and a violation of the contract will be reported.

2.1.2 Higher-Order Contracts

Findler and Felleisen (2002) noted that contracts basing on assertions are not suitable for higher-order functions due to the fact that predicates on functions are not decidable, and they proposed to treat higher-order contracts as values, that are attached to enclosed functions — function contract applied to a function will wrap the enclosed function. Function contracts are composed of other contracts

\[ CD \rightarrow CR, \]

where $CD$ is the domain contract, and $CR$ is the range contract. Function contracts are evaluated before and after the function application, when the values are supplied:

\[
[CD \rightarrow CR] f x \\
\rightarrow [CR] (\lambda x. f ([CD] x))
\]

where $[CD \rightarrow CR]$ is the function contract, $f$ is the enclosed function, and $[CR] (\lambda x. f ([CD] x))$ is a wrapped function that evaluates the contracts during the function application. The domain contract $CD$ is evaluated on the supplied argument $x$, and the
range contract $CR$ is evaluated on the result of applying the function $f$ on $x$. Violation of the domain contract blames the context of the function that supplied the argument $x$, while violation of the range contract blames the function $f$.

### 2.1.3 Blame

Findler and Felleisen (2002) introduced the notion of allocating negative or positive blame upon contract violation, where negative blame corresponded to blaming the context which supplied the arguments, and positive blame corresponded to blaming the function enclosed in the contract.

Since the violation of a higher-order contract might occur in a different section of code, or even in a section of code unavailable to the programmer — a third-party module for example — they concluded that the program should track blame carefully and allocate it to the appropriate module.

In order to track the blame accordingly they decorated contracts with blame labels

$$(CD \to CR)^{p,n} \rightarrow CD^{n,p} \to CR^{p,n}$$

where $p$ is a blame label that corresponds to the positive blame and $n$ is a blame label that corresponds to the negative blame. Upon contract violation, the first blame label is raised.

Revisiting the previous example, when applying the contract $[>10]$ in the presence of some blame labels $r$ and $q$

$[>10]^{q,r} i$

If the value of $i$ is greater than 10, the evaluation will continue uninterrupted, and if the value of $i$ is less or equals than 10, the evaluation will be interrupted and the blame label $q$ will be raised.

Higher-order contracts influenced type systems for mixing statically and dynamically typed code — Gradual typing (J. G. Siek and Taha, 2006) and Hybrid type checking (Flanagan, 2006) — while the absence of blame tracking in those type systems lead to the invention of Blame calculus (Wadler and Findler, 2009).
2.2 Blame Calculus


The parallelism Flanagan (2006) and J. G. Siek and Taha (2006), while neglecting the tracking and allocation of blame, lead to invention of Blame calculus by Wadler and Findler (2009), which was later expanded to include parametric polymorphism (Ahmed et al., 2011).

Blame calculus gives a formal framework for tracking blame and proves the Blame theorem, in which the blame for a failed cast is always allocated to the less precisely typed part — “well typed programs can’t be blamed”.

Blame calculus represents casts as contracts annotated by blame labels used to track blame. Consider the following cast annotated by a blame label, written in the representation used by Ahmed et al. (2011):

$$v : A \Rightarrow^p B$$

for a value $v$, compatible types $A$ and $B$, and a blame label $p$. Assuming that value $v$ belongs to some type $G$, if type $G$ is compatible with the type $B$, denoted by $G \prec B$, the cast will succeed. If the type $G$ is not compatible with the type $B$, denoted by $G \not\prec B$, or if the context containing the cast is incompatible with the type $A$, the cast will fail, and the blame label $p$ will be raised.

Similarly to higher-order contracts, the cast on a function type wraps the function in order to cast the domain and the range of the function

$$f : A \rightarrow B \Rightarrow^p A' \rightarrow B'$$

$$\rightarrow (\lambda x. f(x : A' \Rightarrow^{p/dom} A)) : B \Rightarrow^{p/rng} B'$$

Please note the additional annotation to the blame label $p$, which is not part of the representation by Ahmed et al. (2011). It allows to track blame on both sides of the function cast by specifying the path: Blame label $p/dom$ represents the domain of the function type and blame label $p/rng$ represents the range of
the function type. Explicit tracking of Blame paths is novel, and was shown to the author by Wadler (2014b) when discussing unions for types. Consider the following example of applying function \( pos = \lambda x : \text{Num}. x > 0 \) to untyped value \((1 : \text{Num} \Rightarrow *)\), taken from Ahmed et al. (2011) and written with the explicit tracking of Blame paths.

\[
(pos : \text{Num} \rightarrow \text{Bool} \Rightarrow p \star \rightarrow \star)(1 : \text{Num} \Rightarrow *)
\]

\[
\rightarrow^* pos(1 : \text{Num} \Rightarrow \star \Rightarrow p/\text{dom} \text{Num}) : \text{Bool} \Rightarrow p/\text{rng}
\]

\[
\rightarrow^* pos \ 1 : \text{Bool} \Rightarrow p/\text{rng} \star
\]

\[
\rightarrow^* \text{true} : \text{Bool} \Rightarrow \star
\]

The cast succeeds and returns and returns an untyped value \text{true}. If the function \( pos \) was applied to an argument of incompatible type, for instance \text{false} : \text{Bool} \Rightarrow *, the cast would fail and blame the domain

\[
(pos : \text{Num} \rightarrow \text{Bool} \Rightarrow p \star \rightarrow \star)(\text{false} : \text{Bool} \Rightarrow *)
\]

\[
\rightarrow^* pos(1 : \text{Num} \Rightarrow \star \Rightarrow p/\text{dom} \text{Num}) : \text{Bool} \Rightarrow p/\text{rng}
\]

\[
\rightarrow^* \text{blame} p/\text{dom}
\]

Blame calculus deals with both extending a typed program with portions of untyped code and embedding typed code into an untyped program. As this project deals with type checking untyped code, which belongs to the latter case, type casts will have the following syntax

\[
(v : \star \Rightarrow p A) : B \Rightarrow^q \star
\]

for a value \( v \), the dynamic type \( \star \), types \( A \) and \( B \), and blame labels \( p \) and \( q \). The dynamic type \( \star \) represent a value of any type. As casts are left associative, the above notation is often written with parentheses omitted

\[
v : \star \Rightarrow^p A : B \Rightarrow^q \star
\]

A careful reader might notice that types \( A \) and \( B \) should represent the same type, however I will later show that certain casts require that both types are explicitly tracked. Explicit tracking of both types for casts is novel and was conceived by Wadler (2014a) when he was explaining polymorphic blame calculus to the author.
2.3 JavaScript

Over the past couple of years JavaScript enjoyed a rise in popularity as every modern mobile device capable of accessing a web page and rendering it properly will have some degree of JavaScript support — even specialised devices like the 1st generation Amazon Kindle e-book reader (Amazon, 2007).

Recently Google released Dart, a gradually typed programming language which compiles to JavaScript, with a long term goal of providing an alternative to JavaScript — the experimental Dartium web browser is capable of running Dart natively. In addition, Microsoft released TypeScript, another gradually typed programming language which compiles to JavaScript, with a focus on providing a zero-cost interaction with existing JavaScript code.

2.3.1 TypeScript

TypeScript (2014) adds static type checking using gradual typing system to JavaScript. The main goal of TypeScript is to provide type information for Microsoft’s IntelliSense in order to assist the developer in writing code. It was designed as a lightweight extension with minimal overhead of migrating code from vanilla JavaScript. TypeScript does not provide strict type guarantees and its type checker is deliberately unsound: in order to be more accessible for the developers unfamiliar with type systems it does not enforce consistent subtyping (J. Siek and Taha, 2007) for structural types. Interaction with external untyped libraries uses type definitions specified by the developer and currently there is an actively developed repository with type definitions for around 530 JavaScript libraries (Yankov, 2014).

2.3.2 JavaScript engines

JavaScript implementations or virtual machines are commonly referred to as “engines”. This project will focus on Mozilla SpiderMonkey, the JavaScript engine powering Mozilla Firefox, and Google V8, the JavaScript engine powering Google Chrome and node.js JavaScript runtime environment.

Mozilla SpiderMonkey is the first-ever JavaScript implementation, written by Brendan Eich for Netscape Navigator, later released as open source, and currently
maintained by Mozilla Foundation. SpiderMonkey supports the most complete set of ECMAScript 6 features natively, as compared to other JavaScript engines (Zaytsev, 2012).

**Google V8** is an open source JavaScript engine, initially developed for Google Chrome, that serves as a base for node.js JavaScript runtime, which comes with a large package repository (npm, 2011). As some node.js packages are published with their unit tests, they will be used for evaluating the wrapper.

2.4 Proxies

Proxies (Van Cutsem, 2013) provide general meta-programming for objects and functions in the new JavaScript standard (Ecma, 2014). Proxy’s core concept is that it appears indistinguishable from the enclosed object and that it refers actions performed on the object to a handler which specifies a set of traps (functions which handle intercepted actions). Proxies are natively supported by Mozilla SpiderMonkey and Google V8, but the latter requires enabling them explicitly and requires an external library to support the most recent version of the Proxy API (Van Cutsem, 2014).

2.4.1 Meta-programming

JavaScript extension for Mozilla Firefox 4 introduced meta-programming to JavaScript that allowed for limited *intercession* — changing of semantics from within the language itself. Special method `__noSuchMethod__` defined on an object intercepted calls to methods not defined on the object, but `__noSuchMethod__` was available as a callable and enumerable property, which caused risks for potentially unwanted behaviour: for example, a program iterating over all the object’s properties would iterate over `__noSuchMethod__` as well.

Maes (1987) introduced the concept of separating meta-level functionality from base-level code, also called *stratification*, and Bracha and Ungar (2004) listed it as one of design principles for meta-programming for object-oriented programming languages, since it prevents unwanted access to the meta-level functionality as in the aforementioned example.

ECMAScript 5 (Ecma, 2011) added two relevant constructs to JavaScript: getters and setters for object properties, and tamper-proof objects.
Getters and setters allow for stratified intercession on object properties by providing the JavaScript runtime with functions that intercept retrieving a value from a property and storing a value in a property respectively.

An object becomes tamper-proof by specifying invariants, which make a single property or a whole object immutable, or which prevent adding or removing properties.

### 2.4.2 Direct Proxies

Van Cutsem and Miller (2010) showed that wrapping objects based on getters and setters did not scale and did not account for changes during the object lifetime, as properties added to the object after specifying getters and setters will not be intercepted; they proposed Proxies, influenced by AmbientTalk’s Mirages (Mostinckx et al., 2007), which delegated actions on the proxy object to a separate handler containing traps intercepting those actions, with the exception of object equivalence for security reasons.

Proxies did not access objects directly as handlers enclosed them lexically and in order to address scaling Van Cutsem and Miller (2010) proposed using proxies as handlers for other proxies, resulting in funneling: proxies kept invoking traps up the funnel until reaching a non-proxy handler, and then returned down the funnel. Since a Proxy wrapped around an already existing object, the funneling was guaranteed to finish.

Van Cutsem and Miller (2013) later improved their concept and introduced Direct Proxies, also called Trustworthy Proxies, to ensure that proxy wrappers preserve the invariants specified on the tamper-proof objects. Direct Proxies contain references to the enclosed objects, which allows to verify if the invariants are indeed preserved and simplifies the design of handlers by reducing the need for funneling when scaling: many instances of the same proxy may reuse the same instance of the handler.
Chapter 3

Theory

Firstly, I will present a simplified version of Blame Calculus, that focuses on embedding types into untyped context by inserting casts. This version of Blame Calculus, called Simplified Blame Calculus, will use type casts with explicit tracking of both types participating in the cast (Wadler, 2014a) together with improved blame labels to incorporate blame paths (Wadler, 2014b). Then I will show an extend Simplified Blame Calculus to support arrays, objects. Finally, I will show how to translate this extended Blame Calculus to $\lambda_{JSP}$ a formalised portion of JavaScript based on $\lambda_{JS}$ by Guha et al. (2010) with Proxy representation of Van Cutsem and Miller (2013). I formalise Simplified Blame Calculus and $\lambda_{JSP}$ using Lambda Calculus invented by Church (1940) using the operational semantics as described by Pierce et al. (2014) with evaluation contexts of Felleisen and Hieb (1992).

3.1 Simplified Blame Calculus

There are two possibilities approaches for wrapping objects and arrays:

- eagerly — when each element of an enclosed object is wrapped during the cast,

- lazily — when the whole object is wrapped and elements are wrapped during member access.

Both approaches have their respective advantages and disadvantages: wrapping objects eagerly in a language with side effects might break the semantics of the enclosed object, as members are accessed multiple times. While wrapping objects
Chapter 3. Theory

Syntax

Variables \( x, y \)

Numbers \( n \)

Strings \( s \)

Booleans \( b \) ::= \( \text{true} | \text{false} \)

Constants \( c \) ::= \( b | n | s \)

Base types \( \mathbb{T} \) ::= \{ \ldots \}

Object types \( D, O \) ::= \{ \ldots \}

Types \( A, B, C \) ::= \( \mathbb{T} | A \rightarrow B | \{ x \} | D | * \)

Ground types \( G, H \) ::= \( \mathbb{T} | s \rightarrow s | \{ x \} | \{ x \} \)

Terms \( L, M, N \) ::= \( \mathbb{T} \text{ op}(\mathbb{T} \ldots) | x | \lambda x. M | M \cdot N | M[N] | M[N] = L | \{ \ldots \} | s \rightarrow s | M : s \Rightarrow p : A : B \Rightarrow q \cdot s | \text{ blame p} \)

Environments \( \Gamma \) ::= \( : \mathbb{T} | \Gamma, x : A \)

Values \( v, w \) ::= \( \mathbb{T} \lambda x. M | \{ \ldots : v \ldots \} | (v : D^p, O^q) | (v : \{ A \}^r, \{ B \}^q) \)

Contexts \( E \) ::= \( \{ \} | \mathbb{op}(\mathbb{v} \ldots, E, M, \ldots) | E \cdot N | v \cdot E | \{ x : v \ldots \} : E \cdot M | E[N] | v[E] | E[N] = L \)

\( \text{ Type rules } \)

\[ \Gamma \vdash x : A \quad A \rightarrow B \]

\[ \Gamma \vdash \text{ blame p} : A \quad \Gamma \vdash v : s \]

Compatibility

\[ A \ll A \quad A \ll B \quad B \ll B' \quad A \rightarrow B \ll A' \rightarrow B' \quad \forall s \in \text{ dom}(D) : D[s] \ll O[s] \quad A \ll B \]

Reduction rules

\[ N \rightarrow M \]

\[ E[N] \rightarrow E[M] \]

(\( \lambda x. M \) \( v \) \( \rightarrow \) \( M[x := v] \)) \hspace{1cm} (\text{Beta})

(op(\ldots) \rightarrow \emptyset(\mathbb{op}, \ldots)) \hspace{1cm} (\text{Delta})

\[ \{ \ldots : v \ldots \} [s] \rightarrow v \]

\[ \{ \ldots : v \ldots \} [s] = w \rightarrow \{ \ldots : w \ldots \} \]

(v : * \Rightarrow p : G : G \Rightarrow q \rightarrow s, v : \text{ if } v \in G) \hspace{1cm} (\text{In})

(v : * \Rightarrow p : G : G \Rightarrow q \rightarrow s, v : \text{ if } v \notin G) \hspace{1cm} (\text{Conflict})

(v : * \Rightarrow p : A : B : A' \rightarrow B' \Rightarrow q \rightarrow s, (\lambda x. v : (x : * \Rightarrow \emptyset / \text{ dom } A' : A : \text{ if } x \notin \text{ dom } A' )) : s \Rightarrow p / q \rightarrow B : B' \Rightarrow q / q \rightarrow s) \hspace{1cm} (\text{WrapFun})

(v : * \Rightarrow p : D : O \Rightarrow q \rightarrow s, (v : D^p, O^q)) \hspace{1cm} (\text{WrapObj})

(v : D^p, O^q)[s] \rightarrow (v)[s] : \Rightarrow p / q \rightarrow O[s] \Rightarrow q / q \rightarrow s \hspace{1cm} (\text{WrapObjGet})

(v : D^p, O^q)[s] = w \rightarrow (v)[s] = (w : * \Rightarrow q / q \rightarrow O[s] : D[s] \Rightarrow p / q \rightarrow s)) : \Rightarrow p / q \rightarrow D : O \Rightarrow q \rightarrow s) \hspace{1cm} (\text{WrapObjSet})

(v : * \Rightarrow p : \{ A \}^r, \{ B \}^q) \hspace{1cm} (\text{WrapArr})

(v : \{ A \}^r, \{ B \}^q)[s] \rightarrow (v)[s] : \Rightarrow p / q \rightarrow A : B \Rightarrow q / q \rightarrow s \hspace{1cm} (\text{WrapArrGet})

(v : \{ A \}^r, \{ B \}^q)[s] = w \rightarrow (v)[s] = (w : * \Rightarrow q / q \rightarrow B : A : \Rightarrow p / q \rightarrow s)) : \Rightarrow p / q \rightarrow \{ A \}^r, \{ B \}^q \Rightarrow q \rightarrow s) \hspace{1cm} (\text{WrapArrSet})

(\text{E blame p} \rightarrow \text{ blame p}) \hspace{1cm} (\text{Abort})

Figure 3.1: Simplified Blame Calculus
lazily will not raise blame until a member cast to an incompatible type is accessed and wrapping objects lazily is not space efficient, as the object will be wrapped multiple times.

I chose to wrap objects and arrays lazily, as preserving semantics is crucial for a project that aims at verifying type declarations. The performance of the wrapped code is not addressed by this project as it meant mostly for testing. Still, the space efficiency of wrapping casts is a known problem and it was addressed by J. G. Siek and Wadler (2010)

3.1.1 Blame Paths

Simplified Blame Calculus uses Blame paths (Wadler, 2014b) to improve the allocation of blame in a case of failed cast. In addition to the dom and rng path, the get path is used when accessing object or array members (rules WrapObjGet and WrapArrGet), and the set path is used when updating object or array members (rules WrapObjSet and WrapArrSet).

3.1.2 Objects

I represent the record type as a mapping from string labels to types, \( \{l : A \} \), and array types as the type of the element, \([A]\). For simplicity I do not provide a separate construct for arrays — arrays are records — a simplification used by both Guha et al. (2010) and Van Cutsem and Miller (2013). Array type corresponds to wrapping each element of a record with the same type. There is no support for structural sub typing. In order to check types during member access and update, Simplified Blame Calculus treats values decorated with object types as values.

**Member Access** is handled by the rule WrapObjGet. Consider a wrapped record

\[
b = (\{ \{a' : 1\} : A' : \text{Num}\}^p, \{\{a' : \text{Num}\}^q)\]

Accessing \( b[a'] \) will succeed and return 1

\[
(\{ \{a' : 1\} : A' : \text{Num}\}^p, \{\{a' : \text{Num}\}^q)\}[a']
\]

\[
\rightarrow (\{ \{a' : 1\}[a']\} : A' : \text{Num} \Rightarrow p/\text{get} \text{Num} : \text{Num} \Rightarrow q/\text{get} \text{Num})
\]

\[
\rightarrow 1 : \text{Num} \Rightarrow p/\text{get} \text{Num} \Rightarrow q/\text{get} \text{Num}
\]

\[
\rightarrow 1
\]
If the enclosed the property $a$ of the enclosed record was of a different type, the member access would fail and raise blame $p$/get

\[
\{(a': \text{true}) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q[\text{a}']
\]
\[
\rightarrow\{(a': \text{true})[\text{a}'] : \star \Rightarrow_{p/set} \text{Num} : \text{Num} \Rightarrow_{q/set} \star
\]
\[
\rightarrow\text{true : }\star \Rightarrow_{p/set} \text{Num} : \text{Num} \Rightarrow_{q/set} \star
\]
\[
\rightarrow\text{blame } p/set
\]

**Member Update** is handled by the rule WRAPOBJSET. The additional wrapping at the end of WRAPOBJSET is to ensure that the record returned by the update operation is also wrapped. Consider a wrapped record

\[
b = \{(a' : 1) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q\}
\]

Updating $b'[\text{a}'] = 2$ will succeed and return $\{(a' : 1) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q\}$

\[
\{(a' : 1) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q[\text{a}'] = 2
\]
\[
\rightarrow\{(a' : 1)[\text{a}'] = (2 : \star \Rightarrow_{q/set} \text{Num} : \text{Num} \Rightarrow_{p/set} \star)) : \star \Rightarrow_{p} \{a' : \text{Num}\} : \{a' : \text{Num}\} \Rightarrow_{q} \star
\]
\[
\rightarrow\{(a' : 1)[\text{a}'] = 2 : \star \Rightarrow_{p} \{a' : \text{Num}\} : \{a' : \text{Num}\} \Rightarrow_{q} \star
\]
\[
\rightarrow\text{true : }\star \Rightarrow_{p/set} \text{Num} : \text{Num} \Rightarrow_{q/set} \star
\]
\[
\rightarrow\{a' : 2) : \star \Rightarrow_{p} \{a' : \text{Num}\} : \{a' : \text{Num}\} \Rightarrow_{q} \star
\]
\[
\rightarrow\{(a' : 2) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q\}
\]

If the value to update the property $a'$ was of a type different that Num, then the member update would fail and raise blame $q/set$

\[
\{(a' : 1) : \{a' : \text{Num}\}^p, \{a' : \text{Num}\}^q[\text{a}'] = \text{true}
\]
\[
\rightarrow\{(a' : 1)[\text{a}'] = (\text{true : }\star \Rightarrow_{q/set} \text{Num} : \text{Num} \Rightarrow_{p/set} \star)) : \star \Rightarrow_{p} \{a' : \text{Num}\} : \{a' : \text{Num}\} \Rightarrow_{q} \star
\]
\[
\rightarrow\text{blame } q/set
\]

### 3.1.3 Arrays

Arrays are wrapped similarly to objects, with the main difference, that all elements are wrapped with the same type.

### 3.2 The $\lambda_{JSP}$ Calculus

I modelled a small portion of JavaScript sufficient to represent the constructs implemented in the wrapper as $\lambda_{JSP}$, a JavaScript-like calculus based on $\lambda_{JS}$ by
Guha et al. (2010) — a formal representation that is obtained by transforming JavaScript with a set of desugaring rules. I extended \( \lambda_{JS} \) with the representation of Proxies introduced by Van Cutsem and Miller (2013), and extended the set of desugaring rules with the rules for handling Proxies. The original desugaring rules were tested against test suites for major JavaScript implementations; due to the time constraints new rules are backed by an informal proof only.

### 3.2.1 Desugaring

\( \lambda_{JS} \) is a formal model of a small subset of JavaScript and comes with a set of desugaring rules that allow to transform all of JavaScript code into \( \lambda_{JS} \). Guha et al. noted that, given its nature, the JavaScript language is defined by its implementations; in order to prove the adequacy of \( \lambda_{JS} \), they evaluated their desugaring model on a large portion of the Mozilla SpiderMonkey test suite. In order to back the claim that the desugaring is still adequate after introducing the construct for Proxies, I extend the desugaring rules with the following rule for Proxies:

\[
\text{desugar}[\text{new Proxy}(e_h, e_t)] = \\
\quad \text{let } \text{handler} = \text{desugar}[e_h] \text{ in} \\
\quad \text{let } \text{target} = \text{desugar}[e_t] \text{ in} \\
\quad \text{ref proxy handler of target}
\]

I conjecture that since the added rule targets only Proxies, which were not available in JavaScript when Guha et al. published their paper, it does not affect the remaining of \( \lambda_{JS} \). I also conjecture that it desugars the Proxy constructor to a corresponding \( \lambda_{JS} \) construct.
Syntax

Addresses  $a$

Variables  $x$

Numbers  $n$

Strings  $s$

Booleans  $b$  ::=  $true$ | $false$

Constants  $c$  ::=  $b$ | $n$ | $s$

Heaps  $H$  ::=  $\cdots a : v \cdots$

Terms  $L, M, N$  ::=  $c$ | $op(M \cdots)$ | $x$ | $\lambda x. M$ | $M N$ | $if \ L \ then \ M \ else \ N$

| $\{ \cdots s : M \ \cdots \} | M[N] | M[N] = L | proxy \ M \ of \ N$
| $M = N | ref \ M | deref \ M | throw \ M | typeof \ M$

| $M == N$

Values  $v, w, u$  ::=  $c$ | $\lambda x. M$ | $\{ \cdots s : v \ \cdots \} | proxy \ v \ of \ w | a | err \ v$

Contexts  $E$  ::=  $[\cdot] | \ op(v \cdots, E, M \cdots) | E \ N | v \ E$

| $if \ E \ then \ M \ else \ N | \{s : v \ \cdots s : E \ \cdots s : M\}$
| proxy $E$ of $N | proxy \ v \ of \ E | E = N | v = E$

| $ref \ E | deref \ E | throw \ E | E == N | v == E$

---

Figure 3.2: $\lambda_{JSP}$ Syntax
Reduction rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \rightarrow M$</td>
<td>$H, E[N] \rightarrow H, E[M]$</td>
<td>(Step)</td>
</tr>
<tr>
<td>$(\lambda x. M) v \rightarrow M[x := v]$</td>
<td>(Beta)</td>
<td></td>
</tr>
<tr>
<td>$op(v \cdots) \rightarrow \delta(op, v \cdots)$</td>
<td>(Delta)</td>
<td></td>
</tr>
<tr>
<td>$let x = v in M \overset{\text{def}}{=} (\lambda x. M) v$</td>
<td>(Let)</td>
<td></td>
</tr>
<tr>
<td>$M ; N \overset{\text{def}}{=} (\lambda x. N) M \quad x \notin \text{FV}(N)$</td>
<td>(Seq)</td>
<td></td>
</tr>
<tr>
<td>$v == v \rightarrow true$</td>
<td>(Equals)</td>
<td></td>
</tr>
<tr>
<td>$v == w \rightarrow false$</td>
<td>(NotEquals)</td>
<td></td>
</tr>
<tr>
<td>$\text{typeof } b \rightarrow 'boolean'$</td>
<td>(typeofBool)</td>
<td></td>
</tr>
<tr>
<td>$\text{typeof } s \rightarrow 'string'$</td>
<td>(typeofStr)</td>
<td></td>
</tr>
<tr>
<td>$\text{typeof } n \rightarrow 'number'$</td>
<td>(typeofNum)</td>
<td></td>
</tr>
<tr>
<td>$\text{typeof } (\lambda x. M) \rightarrow 'function'$</td>
<td>(typeofFun)</td>
<td></td>
</tr>
<tr>
<td>$\text{typeof } { \cdots s : M \cdots } \rightarrow 'object'$</td>
<td>(typeofObj)</td>
<td></td>
</tr>
<tr>
<td>if true then $M$ else $N \rightarrow M$</td>
<td>(IfLeft)</td>
<td></td>
</tr>
<tr>
<td>if false then $M$ else $N \rightarrow N$</td>
<td>(IfRight)</td>
<td></td>
</tr>
<tr>
<td>${ \cdots s : v \cdots }[s] \rightarrow v$</td>
<td>(Get)</td>
<td></td>
</tr>
<tr>
<td>${ \cdots s : v \cdots }[s] = w \rightarrow { \cdots s : w \cdots }$</td>
<td>(Set)</td>
<td></td>
</tr>
<tr>
<td>$(\text{proxy } { \cdots &quot;get&quot;: v \cdots } \text{ of } w)[s] \rightarrow v w s$</td>
<td>(ProxyGet)</td>
<td></td>
</tr>
<tr>
<td>$(\text{proxy } { \cdots &quot;set&quot;: v \cdots } \text{ of } w)[s] = u \rightarrow v w s u$</td>
<td>(ProxySet)</td>
<td></td>
</tr>
<tr>
<td>$(\text{proxy } { \cdots &quot;apply&quot;: v \cdots } \text{ of } w)[s] u \rightarrow v w u$</td>
<td>(ProxyApply)</td>
<td></td>
</tr>
<tr>
<td>$a \notin \text{dom}(H) \quad H' = a : v, H$</td>
<td>(Ref)</td>
<td></td>
</tr>
<tr>
<td>$H, E[\text{ref } v] \rightarrow H', E[a]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H, E[\text{deref } a] \rightarrow H, E[H(a)]$</td>
<td>(DeRef)</td>
<td></td>
</tr>
<tr>
<td>$\cdots a : v \cdots, E[a = w] \rightarrow \cdots a : w \cdots, E[a]$</td>
<td>(SetRef)</td>
<td></td>
</tr>
<tr>
<td>throw $v \rightarrow \text{err } v$</td>
<td>(Throw)</td>
<td></td>
</tr>
<tr>
<td>$E[\text{err } v] \rightarrow \text{err } v$, if $E \neq []$</td>
<td>(Abort)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: $\lambda_{JSP}$ Reduction rules
Chapter 4

Implementation

I implemented a command-line tool that allows to generate type-checking wrappers around a JavaScript module using a TypeScript declaration; the tool consists of two major components:

**Wrapper** which contains a `wrap` function which wraps a JavaScript reference with a type, and a Domain Specific Language (DSL) for constructing types for the `wrap` function.

**Parser** which constructs a type, using the DSL provided by the Wrapper, based on a TypeScript ambient declaration read from a TypeScript declaration file.

4.1 Wrapper

4.1.1 Proxy-based Wrappers

Proxy-based wrappers wrap functions and objects transparently, as the proxy allows for preserving the semantics of the enclosed object, aside from the proxy never being equal to the target in terms of pointer equivalence.

Wrappers for basic types are first-order contracts, wrappers for functions and objects are higher-order contracts, since an object is a special case of a closure. Wrappers for `forall` types will additionally use dynamic sealing to ensure relational parametricity: a function accepting a `forall` type will only use information specified in the `forall` type definition.

The advantage of using proxy-based wrappers is that they allow for trapping functionality over objects while preserving the objects’ invariants and type
Proxy support on Google V8 required to recompile the JavaScript runtime environment with special compile flags to ensure that the Proxy support is enabled at all times.

4.1.2 Describing Types

Current JavaScript module loaders bind the external module’s exported namespace into an identifier provided by the developer, therefore avoiding polluting the global namespace. Often, the identifier is named similarly to the module, for instance, when generating the wrappers for external modules the Wrapper is loaded and bound to the identifier `Blame`, and Wrappers exported elements are accessible by `Blame.<element name>`. For simplicity, I will drop the `Blame` prefix and assume that all Wrappers’ exported elements were populated in the global namespace.

DSL exports the following two kinds of elements.

Base Types correspond to the base types of the JavaScript language in a strict sense and no coercions are performed during type checking. Base Types are exported as identifiers beginning with a capital letter:

- `Bool`, which corresponds to the `boolean` JavaScript type. As JavaScript has multiple representations for truth values, this cast will fail if the value is neither exactly `true` nor exactly `false`.

- `Num`, which corresponds to the `number` JavaScript type.

- `Str`, which corresponds to the `string` JavaScript type.

Type Constructors that allow to define more complicated types, each constructor will be described in detail later, when discussing wrapping corresponding JavaScript types. Type constructors exported as identifiers beginning with a small letter:

- `fun` for specifying function types

- `forall` for specifying forall types
Chapter 4. Implementation

- \texttt{arr} for specifying array types
- \texttt{obj} for specifying object types
- \texttt{union} for specifying union types

4.2 Wrap API

The core of the Wrapper is the \texttt{wrap} function which takes a value and a type, generates a set of labels, then returns the value after performing a cast to the provided type. For example, consider wrapping 1, with a Base Type \texttt{Num}

\begin{verbatim}
wrap(1, Num)
\end{verbatim}

Since 1 belongs to the Base Type \texttt{Num} the wrapping succeeds and returns 1. If the value was wrapped with an incompatible type, for example:

\begin{verbatim}
wrap(1, Bool)
\end{verbatim}

the \texttt{wrap} function will fail and report the appropriate blame label.

4.3 Wrapping Functions

Syntax

\begin{verbatim}
fun(required, optional, rest, ret, cons)
\end{verbatim}

Parameters

\begin{verbatim}
required
Array of types for required parameters,
optional
Array of types for optional parameters,
rest
Type for the rest (repeated) parameters,
ret
Type for the value returned by the function,
cons
Type for the object constructed with the \texttt{new} keyword.
\end{verbatim}

---

Figure 4.1: \texttt{fun} Type Constructor signature
As TypeScript’s function declarations support defining optional arguments — arguments that might be omitted during a function call, and rest arguments — repeated arguments that have the same type. In order to support those cases the Type Constructor `fun` takes five arguments: a list of required types, a list of optional type, the rest type, the return type, and the constructor type (fig. 4.1). The last argument is used to wrap constructors, as constructors in JavaScript are functions that are called using the `new` keyword and the newly created instance is not returned from the function, it is bound to the `this` identifier inside the body of the function. For example, consider the following JavaScript function `f`:

```javascript
function f() {
    // Checking if the function is called as a constructor
    if (this instanceof f) {
        this.a = 1;
    }

    return 2;
}
```

which will act differently depending if it is called as a function or as a constructor. If `f` is called as a function, `f()` , it will return 2, whereas if `f` is called as a constructor, `new f()` , it will return a new object `{a: 1}`.

### 4.4 Wrapping Foralls

Forall types make use of tracking both types participating in the cast explicitly (Wadler, 2014a). When an argument is as a type `X` to a forall type, the argument value is sealed under `X`, and when returning a type `X` from a forall type, the value being returned is checked and unsealed — if the value was sealed with the same forall, it is unsealed and returned, otherwise an appropriate blame label is raised.

#### 4.4.1 Dynamic Sealing

In order to seal values securely in JavaScript I use WeakMaps, which are used in JavaScript to store private properties on objects (MDN, 2013).
Chapter 4. Implementation

Syntax
forall(id, type)
Parameters
id
String with the type variable identifier
type
Type containing instances of tyvar with the same id

Syntax
tyvar(id)
Parameters
id
String with the type variable identifier

Figure 4.2: forall and tyvar Type Constructors signature

4.5 Wrapping Arrays and Objects

Syntax
arr(type)
Parameters
type
Type for the elements of the array

Figure 4.3: arr Type Constructor signature

Wrapping Arrays and Objects is performed as described in § 3.1.2.

4.6 Wrapping Unions

The wrapper is capable of wrapping union types, or rather performing unions on higher order contracts. When a side cast participating in an union fails, it reports the blame label back to the union, instead of raising it. When all sides of an union fail, a set of blame labels is reported. This follows the intuition that for a union to fail, all participants from the union should fail first. To the best
Chapter 4. Implementation

Syntax
    obj(properties)
Parameters
    properties
        Mapping from String to Types, describing the types of properties

Figure 4.4: obj Type Constructor signature

Syntax
    union(type...)
Parameters
    type
        Type representing a side of the union

Figure 4.5: union Type Constructor signature

Knowledge of the author the idea of returning a set of blame labels is novel and can be attributed to the author. Unfortunately, this feature was achieved quite late in the project. The main proof the author has to support the claim that it works is a working implementation of the Wrapper that can handle overloaded functions and a working model, written in Haskell by Wadler (2014c) and provided in the appendix, after presenting him with the idea and attempting to simplify the initial implementation.

4.7 Parser

4.7.1 Extracting Type Information

Type information is extracted by instantiating the TypeScript compiler, compiling the relevant type declarations files and processing the symbol tables generated by the compiler. The above approach guarantees that the Parser will successfully parse only declarations that are compatible with TypeScript. It also allows to reuse the type resolving component from the TypeScript compiler, as the declaration syntax is very flexible and allows for splitting a single type declarations into multiple subsequent declarations.
4.8 Wrapper Generation

4.8.1 Dealing with Recursive Types

Parser is capable of dealing with recursive types, a technique common in JavaScript, by creating a list of known named types and generating contracts for them lazily, as they are needed.

4.8.2 Wrapping Modules

Currently Parser supports wrapping external modules for node.js compatible environments. Each module compatible with node.js loading system will define an exported object, called \texttt{module.exports}. Parser generates wrapper code that wraps the \texttt{module.exports} objects, which allows to wrap external libraries without altering the client code — a feature that is extremely useful when reusing unit tests provided with the module.
Chapter 5

Evaluation

The Parser and Wrapper were evaluated using the modules that contained unit tests. As JavaScript modules are published to a public repository (npm, 2011), it was possible to search for modules that had both type declarations in the definitely typed repository (Yankov, 2014) and specified unit tests.

5.1 Acquiring Test Cases

The Parser was evaluated by parsing all available declaration files from the definitely typed repository (Yankov, 2014). Out of 530 modules, 271 modules were successfully parsed. Out of those 271 modules, 107 modules had entries in the npmjs repository that specified unit tests. After downloading modules that specified unit tests, 77 modules passed their unit tests.

5.2 Wrapping Modules

Modules were wrapped by appending the wrapper code to the main module files, as each module specifies a single main file that will be loaded when the module is imported.

5.3 Running Unit Tests

For all of the 77 wrapped modules, the test suites were rerun to determine if the test suites will report any new errors. Afterwards, 36 modules reported additional
errors, meaning that wrapper broke their behaviour, and 41 modules successfully passed their test suites (Table 5.1)

## 5.4 Inspecting Results

Due to the time constraints of the project only limited sample of modules was closely inspected. After inspecting modules closely I managed to fix to determine the cause of failure and fix the declaration. The declarations for the following modules were fixed:

- ansiicolors — more detailed module specification
- deep-freeze — incorrect usage of foralls
- detect-indent — incomplete method signature
- exit — incorrect arguments

Also a common case for failure seemed a small, common quirk for designing JavaScript libraries — certain methods may return null or not return a value at all, a behaviour which is currently impossible to represent as a TypeScript declaration. Although this issue was raised on the TypeScript forum (Polity, 2013), it is unclear when it will be addressed by the developers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Declaration LOC</th>
<th>Module LOC</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansiicolors</td>
<td>9</td>
<td>112</td>
<td>FAIL</td>
</tr>
<tr>
<td>asciify</td>
<td>29</td>
<td>381</td>
<td>FAIL</td>
</tr>
<tr>
<td>assert</td>
<td>62</td>
<td>542</td>
<td>PASS</td>
</tr>
<tr>
<td>assertoin-error</td>
<td>15</td>
<td>191</td>
<td>FAIL</td>
</tr>
<tr>
<td>atmosphere</td>
<td>106</td>
<td>65</td>
<td>PASS</td>
</tr>
<tr>
<td>atpl</td>
<td>20</td>
<td>4611</td>
<td>FAIL</td>
</tr>
<tr>
<td>big.js</td>
<td>199</td>
<td>35681</td>
<td>PASS</td>
</tr>
<tr>
<td>bootbox</td>
<td>47</td>
<td>2682</td>
<td>FAIL</td>
</tr>
<tr>
<td>chai</td>
<td>283</td>
<td>8406</td>
<td>FAIL</td>
</tr>
<tr>
<td>chai-jquery</td>
<td>37</td>
<td>887</td>
<td>FAIL</td>
</tr>
<tr>
<td>cheerio</td>
<td>215</td>
<td>10652</td>
<td>FAIL</td>
</tr>
<tr>
<td>clone</td>
<td>17</td>
<td>282</td>
<td>FAIL</td>
</tr>
</tbody>
</table>

Table 5.1: Results of running 77 wrapped modules
<table>
<thead>
<tr>
<th>Name</th>
<th>Declaration LOC</th>
<th>Module LOC</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>colors.js</td>
<td>26</td>
<td>482</td>
<td>PASS</td>
</tr>
<tr>
<td>commander</td>
<td>227</td>
<td>716</td>
<td>FAIL</td>
</tr>
<tr>
<td>configstore</td>
<td>38</td>
<td>125</td>
<td>PASS</td>
</tr>
<tr>
<td>convert-source-map</td>
<td>26</td>
<td>391</td>
<td>FAIL</td>
</tr>
<tr>
<td>datejs</td>
<td>136</td>
<td>555216</td>
<td>FAIL</td>
</tr>
<tr>
<td>deep-freeze</td>
<td>9</td>
<td>32</td>
<td>FAIL</td>
</tr>
<tr>
<td>detect-indent</td>
<td>9</td>
<td>236</td>
<td>FAIL</td>
</tr>
<tr>
<td>diff</td>
<td>57</td>
<td>1205</td>
<td>PASS</td>
</tr>
<tr>
<td>dot</td>
<td>49</td>
<td>1059</td>
<td>PASS</td>
</tr>
<tr>
<td>dropzone</td>
<td>79</td>
<td>10713</td>
<td>PASS</td>
</tr>
<tr>
<td>dustjs-linkedin</td>
<td>167</td>
<td>13351</td>
<td>FAIL</td>
</tr>
<tr>
<td>es6-promises</td>
<td>8</td>
<td>310</td>
<td>PASS</td>
</tr>
<tr>
<td>esprima</td>
<td>276</td>
<td>90078</td>
<td>FAIL</td>
</tr>
<tr>
<td>exit</td>
<td>9</td>
<td>190</td>
<td>FAIL</td>
</tr>
<tr>
<td>expectations</td>
<td>60</td>
<td>783</td>
<td>PASS</td>
</tr>
<tr>
<td>express-validator</td>
<td>161</td>
<td>544</td>
<td>FAIL</td>
</tr>
<tr>
<td>fibers</td>
<td>24</td>
<td>2281</td>
<td>PASS</td>
</tr>
<tr>
<td>fuse</td>
<td>28</td>
<td>921</td>
<td>FAIL</td>
</tr>
<tr>
<td>geojson</td>
<td>126</td>
<td>433</td>
<td>PASS</td>
</tr>
<tr>
<td>googlemaps</td>
<td>1599</td>
<td>783</td>
<td>PASS</td>
</tr>
<tr>
<td>hammerjs</td>
<td>142</td>
<td>13645</td>
<td>FAIL</td>
</tr>
<tr>
<td>handlebars</td>
<td>68</td>
<td>12995</td>
<td>FAIL</td>
</tr>
<tr>
<td>hashmap</td>
<td>71</td>
<td>344</td>
<td>FAIL</td>
</tr>
<tr>
<td>http-string-parser</td>
<td>38</td>
<td>352</td>
<td>PASS</td>
</tr>
<tr>
<td>jquery.cookie</td>
<td>32</td>
<td>507</td>
<td>FAIL</td>
</tr>
<tr>
<td>jquery.payment</td>
<td>25</td>
<td>1112</td>
<td>PASS</td>
</tr>
<tr>
<td>js-git</td>
<td>188</td>
<td>3575</td>
<td>PASS</td>
</tr>
<tr>
<td>jsesc</td>
<td>25</td>
<td>1239</td>
<td>FAIL</td>
</tr>
<tr>
<td>json-pointer</td>
<td>83</td>
<td>335</td>
<td>FAIL</td>
</tr>
<tr>
<td>jsrender</td>
<td>56</td>
<td>450</td>
<td>PASS</td>
</tr>
<tr>
<td>jszip</td>
<td>182</td>
<td>10038</td>
<td>PASS</td>
</tr>
<tr>
<td>karma-jasmine</td>
<td>9</td>
<td>2374</td>
<td>PASS</td>
</tr>
<tr>
<td>less</td>
<td>552</td>
<td>196919</td>
<td>FAIL</td>
</tr>
</tbody>
</table>

Table 5.1: Results of running 77 wrapped modules
<table>
<thead>
<tr>
<th>Name</th>
<th>Declaration LOC</th>
<th>Module LOC</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>libxmljs</td>
<td>110</td>
<td>168049</td>
<td>FAIL</td>
</tr>
<tr>
<td>linq</td>
<td>222</td>
<td>5563</td>
<td>PASS</td>
</tr>
<tr>
<td>long</td>
<td>80</td>
<td>1492</td>
<td>PASS</td>
</tr>
<tr>
<td>marked</td>
<td>120</td>
<td>1402</td>
<td>FAIL</td>
</tr>
<tr>
<td>minimist</td>
<td>26</td>
<td>678</td>
<td>FAIL</td>
</tr>
<tr>
<td>mixpanel</td>
<td>72</td>
<td>861</td>
<td>PASS</td>
</tr>
<tr>
<td>mixto</td>
<td>16</td>
<td>174</td>
<td>PASS</td>
</tr>
<tr>
<td>mocha</td>
<td>110</td>
<td>8794</td>
<td>PASS</td>
</tr>
<tr>
<td>moment</td>
<td>363</td>
<td>50546</td>
<td>FAIL</td>
</tr>
<tr>
<td>msgpack</td>
<td>83</td>
<td>20747</td>
<td>PASS</td>
</tr>
<tr>
<td>minimist</td>
<td>54</td>
<td>2032</td>
<td>PASS</td>
</tr>
<tr>
<td>nodemailer</td>
<td>106</td>
<td>1015</td>
<td>FAIL</td>
</tr>
<tr>
<td>open</td>
<td>9</td>
<td>611</td>
<td>FAIL</td>
</tr>
<tr>
<td>pegjs</td>
<td>19</td>
<td>12847</td>
<td>PASS</td>
</tr>
<tr>
<td>phonegap</td>
<td>627</td>
<td>6747</td>
<td>PASS</td>
</tr>
<tr>
<td>pixi</td>
<td>447</td>
<td>16563</td>
<td>PASS</td>
</tr>
<tr>
<td>platform.js</td>
<td>30</td>
<td>7114</td>
<td>FAIL</td>
</tr>
<tr>
<td>power-assert</td>
<td>62</td>
<td>12420</td>
<td>PASS</td>
</tr>
<tr>
<td>pubsubjs</td>
<td>30</td>
<td>517</td>
<td>PASS</td>
</tr>
<tr>
<td>qunit</td>
<td>723</td>
<td>941</td>
<td>PASS</td>
</tr>
<tr>
<td>readdir-stream</td>
<td>11</td>
<td>117</td>
<td>PASS</td>
</tr>
<tr>
<td>rimraf</td>
<td>16</td>
<td>285</td>
<td>PASS</td>
</tr>
<tr>
<td>riotjs</td>
<td>12</td>
<td>557</td>
<td>PASS</td>
</tr>
<tr>
<td>scroller</td>
<td>50</td>
<td>1199</td>
<td>PASS</td>
</tr>
<tr>
<td>semver</td>
<td>75</td>
<td>2432</td>
<td>FAIL</td>
</tr>
<tr>
<td>should</td>
<td>118</td>
<td>4078</td>
<td>FAIL</td>
</tr>
<tr>
<td>sjcl</td>
<td>556</td>
<td>28038</td>
<td>FAIL</td>
</tr>
<tr>
<td>stripe</td>
<td>61</td>
<td>3200</td>
<td>PASS</td>
</tr>
<tr>
<td>type-detect</td>
<td>18</td>
<td>522</td>
<td>PASS</td>
</tr>
<tr>
<td>update-notifier</td>
<td>36</td>
<td>147</td>
<td>PASS</td>
</tr>
<tr>
<td>uri-templates</td>
<td>18</td>
<td>3756</td>
<td>FAIL</td>
</tr>
<tr>
<td>regexp</td>
<td>62</td>
<td>14190</td>
<td>PASS</td>
</tr>
</tbody>
</table>

Table 5.1: Results of running 77 wrapped modules
5.5 Measuring modules Lines of Code

Each module and its declaration file had their Lines of Code (LOC) measured using the cloc utility (Danial, 2014). Some modules showed an extremely large number of LOC, which is probably due to the cloc utility also counting the libraries used by the module or extensive large test cases.
Chapter 6

Conclusion

The results show that the type checking tools is capable of detecting incorrectly specified type declarations and improve their quality by enforcing that the declaration strictly conforms to the behaviour of the module.

However, there are certain behaviours that cannot be described using TypeScript type declarations and are currently discussed by the TypeScript community.

Finally, this project provides a novel solution to solve the problem of dealing with unions of higher order contracts.
Appendix A

Working model for union contracts in Haskell

module GradualUnion where
import Control.Monad.State
import Control.Monad.Except
import Control.Monad.Identity
import Data.Maybe
import Data.Either
-- import Data.Map

-- maps

type Map k a = [(k,a)]

(!) :: Eq k => Map k a -> k -> a
m ! k = fromJust (lookup k m)

empty :: Map k a
empty = []

insert :: Eq k => k -> a -> Map k a -> Map k a
insert k a [] = [(k,a)]
insert k a ((k',a') : m) | k == k' = (k,a) : m
| otherwise = (k',a') : insert k a m

-- trees

data Tree a = Nil
  | Leaf a
  | Node (Tree a) (Tree a)
  deriving (Eq,Show)

nil :: Tree a
nil = Nil

add :: [Bool] -> a -> Tree a -> Tree a
add bs a (Node l r) = Node (add bs a l) r
add (True:bs) a (Node 1 r) = Node (add bs a 1) r
add (False:bs) a (Node 1 r) = Node 1 (add bs a r)

complete :: Tree a -> Bool
complete Nil = False
complete (Leaf a) = True
complete (Node 1 r) = complete 1 && complete r
Appendix A. Working model for union contracts in Haskell

traverse :: Tree a -> [a]
traverse (Leaf a) = [a]
traverse (Node l r) = traverse l ++ traverse r

-- paths

data Label = L String
  deriving Eq

instance Show Label where
  show (L s) = s

data Path = Lab Label
  | Dom Path
  | Rng Path
  | Lft Path
  | Rgt Path
  deriving Eq

instance Show Path where
  show (Lab l) = show l
  show (Dom p) = show p ++ "/dom" 
  show (Rng p) = show p ++ "/rng" 
  show (Lft p) = show p ++ "/lft" 
  show (Rgt p) = show p ++ "/rgt"

label :: Path -> Label
label (Lab l) = 1
label (Dom p) = label p
label (Rng p) = label p
label (Lft p) = label p
label (Rgt p) = label p

route :: Path -> [Bool]
route (Lab l) = [ ]
route (Dom p) = route p
route (Rng p) = route p
route (Lft p) = route p ++ [True]
route (Rgt p) = route p ++ [False]

lab :: String -> Path
lab s = Lab (L s)

-- computational effects

type St = Map Label (Tree Path)
type Err = [Path] -- all with the same label
type Eff = ExceptT Err (StateT St Identity)

st0 :: St
st0 = empty

runEff :: Eff a -> (Either Err a, St)
runEff m = runIdentity (runStateT (runExceptT m) st0)

blame :: Path -> Eff ()
blame p = do let l = label p
    m <- get
    let t = fromMaybe nil (lookup l m)
    let t' = add (route p) p t
    let m' = insert l t' m
    put m'
    if complete t' then throwError (traverse t') else return ()

-- untyped lambda calculus

type Id = String
Appendix A. Working model for union contracts in Haskell

data Ground = GInt | GBool | GFun
  deriving (Eq, Show)

data Type = TInt | TBool | Type -> Type | TAny | TZero | Type :+: Type
  deriving Show

data Term = Con Val | Var Id | Lam Id Term | App Term Term | Is Term Ground | If Term Term Term | Cast Term Path Type
  deriving Show

data Val = VInt Int | VBool Bool | VFun (Val -> Comp)

class Inj a where
  inj :: a -> Val

class Prj a where
  prj :: Val -> Eff a

instance Inj Int where
  inj i = VInt i

instance Prj Int where
  prj (VInt i) = return i
  prj v = throwError [lab ("not VInt: "+show v)]

instance Inj Bool where
  inj b = VBool b

instance Prj Bool where
  prj (VBool b) = return b
  prj v = throwError [lab ("not VBool: "+show v)]

instance (Prj a, Inj b) => Inj (a -> b) where
  inj f = VFun (\x -> do v <- prj x
                           return (inj (f v)))

eval :: Term -> Map Id Val -> Op
eval (Con v) r = return v
eval (Var x) r = return (r ! x)
eval (App m n) r = return (VFun (\v -> eval n (insert x v r)))
eval (Lam x m) r = do f <- eval 1 r; v <- eval m r; 'app' v
eval (Is m g) r = do v <- eval m r; return (inj (ground v == g))
eval (If l m n) r = do v <- eval 1 r; b <- prj v; if b then eval m r else eval n r
eval (Cast m p a) r = do v <- eval m r; cast v p a

app :: Val -> (Val -> Op)
Appendix A. Working model for union contracts in Haskell

app (VPan f) v = f v
con :: Inj a => a -> Term
con v = Con (inj v)
cast :: Val -> Path -> Type -> Type

-- case ground v of
GInt -> return v
GBool -> return v
GPer -> return (VPan \x -> do w <- cast x (GDom p) (dom a)
y <- v 'app' w
case y (RGH p) (rng a))

compatible :: Ground -> Path -> Type -> Eff ()

-- compatible g p TInt = if g == GInt then return () else blame p
compatible g p TBool = if g == GBool then return () else blame p
compatible g p TAny = return ()

compatible g p TZero = blame p
compatible g p (a :+: b) = do compatible g (LInt p) a
compatible g (RInt p) b

dom :: Type -> Type
dom TInt = TZero
dom TBool = TZero
dom (a :+: b) = a
dom TAny = TAny
dom TZero = TZero
dom (a :+: b) = dom a :+: dom b

rng :: Type -> Type
rng TInt = TZero
rng TBool = TZero
rng (a :+: b) = b
rng TAny = TAny
rng TZero = TZero
rng (a :+: b) = rng a :+: rng b

-- Tests

plus = con (1 :+: 1 :: Int -> Int -> Int)
one = con (1 :+: 1 :: Int)
three = con (3 :+: 3 :: Int)
seven = con (7 :+: 7 :: Int)
true = con TTrue
twice = (Lam "x" (plus (App Var (Var "x") (App Var (Var "x")))))
incr = (Lam "y" (plus (App Var (Var "y") (App Var (Var "y")))))
g3 = (Lam "y" (Var "y" (App three)))
gt = (Lam (Var "y" (Var "y" (App true)))
kseven = (Lam (Var ",", seven))
ktrue = (Lam (Var ",", true))
b = (Lam (Var "x" (If (Is (Var "x") GInt) (incr (App Var (Var "x") (Var "x"))))))
t0 = twice (App three)
t1 = Cast twice (lab "t1") (TInt -> TInt) 'App' three
t2 = Cast twice (lab "t2") (TInt -> TInt) 'App' true
t3 = Cast twice (lab "t3") (TInt -> TBool) 'App' three
t4 = Cast g3 (lab "t4") (TInt -> TInt) 'App' incr

-- t5 = Cast g3 'App' (Cast incr (lab "t6") (TInt -> TInt))
t7 = gt 'App' (Cast incr (lab "t7") (TInt -> TInt))
t8 = (Cast kseven (lab "t8") (TInt -> TInt) :+: (TAny -> TBool))) 'App' three
t9 = (Cast ktrue (lab "t9") (TInt -> TInt) :+: (TAny -> TBool))) 'App' true
t10 = (Cast ktrue (lab "t10") (TInt -> TInt) :+: (TAny -> TBool))) 'App' three
t11 = (Cast ktrue (lab "t11") (TInt -> TInt) :+: (TAny -> TBool))) 'App' true
t12 = (Cast h (lab "t12") (TInt -> TInt) :+: (TBool -> TBool))) 'App' three
t13 = (Cast h (lab "t13") (TInt -> TInt) :+: (TBool -> TBool))) 'App' true
t14 = (Cast h (lab "t14") (TInt -> TInt) :+: (TBool -> TBool))) 'App' incr
t15 = (Cast incr (lab "t15") (TInt -> TInt) :+: (TBool -> TBool)) :+: (TBool -> TInt)) 'App' three
t16 = (Cast incr (lab "t16") (TInt -> TInt) :+: (TBool -> TBool) :+: (TBool -> TInt)))
Appendix A. Working model for union contracts in Haskell

main =
  check t a = show (test t) == a

-- Flattening types with unions to lists of simple types

simple :: Type -> Bool
simple TInt = True
simple TBool = True
simple (a :+: b) = simple a && simple b
simple TAny = True
simple TZero = False
simple (a :+: b) = False

flatten :: Type -> [Type]
flatten TInt = [TInt]
flatten TBool = [TBool]
flatten (a :+: b) = [a => t | s <- flatten a, t <= flatten b]
flatten TAny = [TAny]
flatten TZero = []
flatten (a :+: b) = flatten a ++ flatten b

{-
  listcast :: Path -> Val -> (Index, Type) -> Cmp
  listcast p v as =
    sequence [simplecast i p v a | (i, a) <- zip [0..] as]
  do sequence [simplecompat i p (ground v) a | (i, a) <- zip [0..] as]
      case ground v of
        GInt -> return v
        GBool -> return v
-}
Appendix A. Working model for union contracts in Haskell

\[
G\text{Fun} \rightarrow \text{return} (Y\text{Fun} (\lambda x \rightarrow \text{do } w \leftarrow \text{cast } x \{\text{Dom } p\} \{\text{dom } a\})
\]
\[
y \leftarrow v \ 'app' \ w
\]
\[
\text{cast } y \{\text{Rng } p\} \{\text{rng } a\}})
\]

\text{simplecompat :: } \text{Int} \rightarrow \text{Path} \rightarrow \text{Ground} \rightarrow \text{Type} \rightarrow \text{Eff} ()
\]
\text{simplecompat _ GInt TInt } = \text{return ()}
\text{simplecompat _ GBool TBool } = \text{return ()}
\text{simplecompat _ GFun (_ :- > _)} = \text{return ()}
\text{simplecompat _ _ TAny } = \text{return ()}
\text{simplecompat i _ _ } = \text{simpleblame i p}
\]
Guha, Arjun, Claudiu Saftoiu and Shriram Krishnamurthi (2010). ‘The essence
of JavaScript’. In: ECOOP 2010–Object-Oriented Programming. Springer,
pp. 126–150.


Maes, Pattie (1987). ‘Concepts and experiments in computational reflection’. In:


‘Mirages: Behavioral intercession in a mirror-based architecture’. In: Proceed-
ings of the 2007 symposium on Dynamic languages. ACM, pp. 89–100.

Mozilla Developer Network (2013). Private Properties using WeakMaps. URL:


Parnas, David Lorge (1972). ‘A technique for software module specification with

Pierce, Benjamin C, Chris Casinghino, Marco Gaboardi, Michael Greenberg,
Sjoberg, Cătălin Hrițcu, Vilhelm Sjöberg and Brent Yorgey (2014). Software
foundations — version 3.1.


Reynolds, John C (1985). ‘Three approaches to type structure’. In: Mathematical

Rosenblum, David S. (1995). ‘A practical approach to programming with asser-


