CHALMERS

GÖTEBORGS UNIVERSITET

Gradual Typing for a More Pure JavaScript

Bachelor's Thesis in Computer Science and Engineering

JAKOB ERLANDSSON
ERIK NYGREN
OSKAR VIGREN
ANTON WESTBERG

Department of Computer Science and Engineering
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Abstract

Dynamically typed languages have surged in popularity in recent years, owing to their flexibility and ease of use. However, for projects of a certain size dynamic typing can cause problems of maintainability as refactoring becomes increasingly difficult. One proposed solution is the use of gradual type systems, where static type annotations are optional. This results in providing the best of both worlds. The purpose of this project is to create a gradual type system on top of JavaScript. Another goal is to explore the possibility of making guarantees about function purity and immutability using the type system. The types and their relations are defined and a basic type checker is implemented to confirm the ideas. Extending type systems to be aware of side effects makes it easier to write safer software. It is concluded that all of this is possible and reasonable to do in JavaScript.

Sammanfattning

Dynamiskt typade programmeringsspråk har ökat kraftigt i popularitet de senaste åren tack vare deras flexibilitet och användbarhet. För projekt av en viss storlek kan dock dynamisk typning skapa underhållsproblem då omstrukturering av kod blir allt svårare. En föreslagen lösning är användande av så kallad gradvis typning där statiskt typad annotering är frivillig, vilket i teorin fångar det bästa av två världar. Syftet med det här projektet är att skapa ett gradvist typsystem ovanpå Javascript. Ett ytterligare mål är att undersöka möjligheten att ge garantier om renhet och muterbarhet genom typsystemet. Typerna och deras relationer är definierade och en enkel typkontrollerare implementeras för att bekräfta idéerna. Utökande av typsystem för att ta hänsyn till sidoeffekter förenklar utvecklingen av säker mjukvara. Slutligen visas att allt detta är möjligt och rimligt att göra i Javascript.

Acknowledgements

We would like to thank our supervisor Ulf Norell at Chalmers University of Technology for taking his time and assisting us throughout the course of the project.

Contents

1 Introduction

In programming languages types are used to classify data into categories. Within these categories operations are limited to those that are defined for the given type. For instance, a multiplication operation might make sense for integers but not for text strings. Performing an illegal operation could set the program into an undefined state, causing it to crash or to corrupt data. Type checkers can find such operations and warn developers of the potential consequences. See Listing 1 for an example of a potential type error in a JavaScript program.

```
//:: string -> string
function firstChar(text) {
    return text.charAt(0)
}
firstChar(100)
```

Listing 1: The *charAt* method fetches the n:th character of a string. Without static type checking the evaluation would result in a run-time exception.

1.1 Type Systems

The way in which types interact with one another is defined as a set of rules called the type system. Type systems traditionally mostly, or exclusively, adhere to one of two techniques: dynamic typing or static typing.

Dynamic typing associates types with run-time values. Here, a variable can hold values of any type and change their type during the course of execution. Type correctness is checked at run-time, when the operations are executed, thus, possible type errors will result in run-time exceptions.

Types in static type systems are associated with bindings. A variable will be assigned a static type and can only ever contain values of that same type. This means that type correctness can be checked before the program is run, avoiding run-time errors (note that a program need not necessarily be compiled to be statically type checked).

Some statically typed languages still tag run-time values with type information. For example of usage, see reflection in C# [1].

The respective advantages of static and dynamic type checking is still widely debated, and is partly a matter of preference. Generally speaking, dynamically typed languages are considered to be more flexible. The flexibility enables faster development time early on in projects, making them suitable for rapid prototyping. On the other hand proponents of static typing point to the extra safety: the assurance that a significant category of errors are ruled out completely.

A more recent development is the introduction of gradual typing, a variety of type system that tries to unify the best parts of dynamic and static typing by allowing programmers to move freely between the two [2]. The idea is that you

can start by writing dynamic code, and then gradually introduce static types over time as the problem domain becomes more clear.

A common origin of gradually typed languages is the introduction of optional static types to an existing dynamic language. This is the case with TypeScript for JavaScript and Typed Racket for Racket.

1.2 Further Research

Designing a type system may result in economic and humanitarian benefits. A more permissive and easy to understand type system reduces program development time, while a strict and extensive type system may prevent catastrophic bugs. Though these are not each others negations, type systems usually need to balance between the two.

JavaScript is currently one of the most used programming languages and is utilized on a substantial part of the modern web. Given this popularity, and JavaScript's notoriously hard to use dynamic type system, improving correctness and safety for JavaScript is especially interesting.

While there are existing well-developed type system extensions for JavaScript, like the above mentioned TypeScript, none have implemented guarantees about the purity of functions. Such guarantees could for example ensure programmers that they need not worry about hidden side effects in well typed, pure functions. See Listing 2 for an example of such a guarantee in action.

```
//:: number, number -> number
function pureAdd(a, b){
  launchMissiles(); // Type check error
  return a + b;
}
```

Listing 2: A pure function should not affect the "outside world". In this slightly contrived example, the pure addition function can not also launch missiles.

1.3 Purpose and Goals

This report presents a gradual type system for JavaScript, introducing the ability to statically verify code with type annotations. The type system use the same primitive types as the existing dynamic type system but with stricter rules.

Furthermore, this report studies verification of purity of JavaScript functions, as well as incorporating this into the type system.

A partial implementation of the type system, a type checker, is presented as well[1]. It is designed to be easy to introduce to any project without refactoring the project's existing code base.

[1]The implementation is publicly available at `https://github.com/Kirens/typkoll/tree/report-release`

In addition, this report aims to provide insight into the thought process behind developing the type system and type checker. Moreover, problems that arose during the development process and how one might go about solving them are also presented.

2 Theory

A statically typed language is checked before execution, meaning that some guarantees can be made about how the code will behave when run. In contrast, a dynamically typed language does not have these types of checks, and type errors will result in run-time exceptions. Gradual type systems try to combine these two paradigms by making static types optional.

In this section we will expand on the theory of gradual type systems and concepts related to implementing one with purity guarantees in JavaScript.

2.1 Gradual Typing

In 2006 Jeremy Siek together with Wahid Taha laid the foundation of what they called gradual typing [2]. This formalized a way for dynamically typed code to interact with statically typed code. In the gradual type system a *dynamic type* (`?`) is introduced and any code that is not explicitly annotated is in fact not untyped but is implicitly given the dynamic type. In addition, a *consistency relation* ($\sim$) between two types is defined. The consistency relation denotes that two types can be implicitly coerced into one another. Siek and Taha defines the following rules:

1. For any type T, we have both $\texttt{?} \sim T$ and $T \sim \texttt{?}$.
2. For any basic type B, such as `int`, we have $B \sim B$.
3. A tuple type $T_1 * T_2$ is consistent with another tuple type $S_1 * S_2$ if $T_1 \sim S_1$ and $T_2 \sim S_2$. This rule generalizes in a straightforward way to tuples of arbitrary size.
4. A function type $\texttt{fun}(T_1, \ldots, T_n, R)$ (the $T_1, \ldots, T_n$ are the parameter types and R is the return type) is consistent with another function type $\texttt{fun}(S_1, \ldots, S_n, U)$ if $T_1 \sim S_1, \ldots, T_n \sim S_n$ and $R \sim U$.

Since $\sim$ accepts conversions between `?` and all other types, its possible to smoothly transition between annotated and unannotated parts of the code. Consequently, this makes it possible to gradually add static types. Furthermore, it enables the programmer to explicitly convert from one type to another through the dynamic type, even though the actual value isn't changed. For example, a cat of type `Cat` can be coerced to `Animal` in two steps, since $\texttt{Cat} \sim \texttt{?}$ and $\texttt{?} \sim \texttt{Animal}$. Caution is needed when using this feature as the same reasoning can be applied to $\texttt{string} \sim \texttt{?}$ and $\texttt{?} \sim \texttt{number}$. As a result, the static type checking is circumvented and possible type errors must be caught by the dynamic type checker, potentially resulting in run-time errors.

2.2 Blame Calculus

One rule used by gradual type systems to determine what code is responsible for errors during run-time is described by Wadler and Findler as blame [3]. According to a set contract, blame determines which part of the program is

responsible for throwing an error. This might seem trivial when converting a simple type, but a function which is cast to another function with the dynamic type as its domain does a run-time cast on its argument. Wadler and Findler call this system blame calculus. The main point of defining the blame calculus is to assure that a well typed program never can be blamed. In essence, if a type error occurs, it is always the fault of the dynamically typed part.

2.3 The JavaScript Type System

Javascript being a dynamically typed language means no type checks are done statically. Instead, the interpreter throws the necessary errors at run-time. There are tools built which help the developer spot potential errors while writing the code. However, they can only do so much as some expressions need to be evaluated in order for a computer to decide whether or not they will fail.

JavaScript has seven built in data types: *boolean*, *number*, *string*, *symbol*, *null*, *undefined* and *object*. The first six of these are called primitive types and are immutable. The seventh type, object, can be thought of as a mapping of property names to values. In contrast to values of other types, an instantiated object can be freely mutated, meaning properties can change types, be removed or added. Functions are formally a callable object and the arguments are therefore not part of the type. While this makes them flexible, it might also make them unsafe since there is no guarantee the function's arguments are of a certain type. Variables and properties that have not yet been given any value are of the type undefined.

A common source of confusion for new JavaScript users is the fact that it uses a lot of implicit conversions between types. This is sometimes referred to as *weak typing*, though there is no universally agreed upon definition. This means that the interpreter will often try to convert between types in favour of run-time errors. For example, evaluating `'1' + 1`, the string `'1'` plus the number `1`, would in many other languages result in a type error since numbers and strings are different types. However, in JavaScript the result is the string `'11'`, since the `+` operator will do an automatic conversion. Similarly, `'12'/2` evaluates to the number `6`. The interpreter will also implicitly convert all values to boolean in certain context, like in the test of an if statement. It is said that values are either *truthy* or *falsy*. This can sometimes be handy, for example checking that variable `x` is defined with `if(x)`. However, the coercion rules are not always intuitive, notably the empty string being falsy where as an empty list being truthy.

2.4 Functional Purity

Though there is no universally agreed upon definition of purity in programming languages, in this report functions will be considered pure if they are referentially transparent. This means that a function call may be replaced with its result without affecting the behaviour of the program. This definition seems to be the consensus when reading articles, blog posts and other literature on this subject. Consequently, pure functions never have any side effects and for any

given input they will always produce the same output independent of variables changing outside the function's local scope.

Annotating a function as pure would assure the programmer that it can be used anywhere and behave as expected without unexpected side effects. As a result, the code may be reasoned about in isolation and is easier to understand.

Gifford and Lucassen define what they call a *fluent language*, as a mix of functional and imperative programming [4]. Fluent programming facilitates the aforementioned consequences of purity, but also introduce a hierarchy of purity with decreasingly relaxed rules.

2.5 Data Immutability

Another distinctive feature of functional programming languages is that data is predominantly immutable. In other words once a value is assigned to a variable it can never be changed. As a result, reasoning about the program is easier since the programmer does not have to worry about data changing in unpredictable ways. In JavaScript, values of primitive types are already immutable while objects are all by default mutable.

3 Methodology: Design Process

The type system and a corresponding type checker were implemented in parallel to ensure correct functionality of both at all time. The following sections will discuss the external libraries and tools used to develop the type checker prototype, as well as the work methodology used.

3.1 Tools Used in the Implementation

In order to speed up development of the product, a few pre-built tools were used to save time that would have otherwise gone into developing similar tools. These tools helped in getting the process of developing the type system started quicker as well as ensuring that these vital parts of the program worked flawlessly.

3.1.1 Esprima, the JavaScript Parser

Esprima is a library for parsing JavaScript code into an abstract syntax tree (AST), with nodes for all statements, expressions and other language constructs [5]. It is widely used as a base for building JavaScript tooling. As further discussed in subsection 4.1 *Overview of the Type Checker* all type annotations are written as regular comments and are as such valid JavaScript that Esprima can parse.

3.1.2 Jison, the Type Annotation Parser Generator

To parse the type annotations the parser generator Jison was used. Jison is a JavaScript implementation of the popular Bison tool [6]. It takes a context free grammar as input and generates JavaScript code capable of parsing the language described by the grammar. A grammar was created, defining the various constructs of the type system. The code generated from this grammar can parse the type annotation and definition comments in the AST produced by Esprima.

3.2 Test-Driven Development

When developing the type checker the Test-driven development (TDD) methodology was used. TDD is a strategy for improving software quality by emphasising extensive unit testing. In practice, this means that before implementing a new feature or when fixing a bug, a failing test using the new, non-existent, functionality is written. After that the feature is implemented, making the test pass. As a result the code can be refactored, as long as the tests keep passing. At last new failing tests are written for the next piece of functionality, starting the cycle again.

The benefit of this style of development became clear during the project. Owing to that no lines of production code is written without a test, there was a high degree of confidence in the test suite. One could be relatively sure that as long as

all the tests passed, no errors had been introduced. This allowed refactorization that was relatively painless and safe, an important benefit in a project that was highly malleable and iterative.

3.3 Division of Work

Since the product consists of two main parts, the type system and a type checker for that system, it made sense to parallelise the work between team members. However, as both parts rely on the functionality of the other the team had to work carefully and document in great detail how the work was carried out as to not end up with incompatible parts.

4 Results: Design and Implementation

The type system allows static type checking by annotating declarations of variables and functions with types. The code is then analyzed to find any mismatches, where for example values of a certain type are assigned to a variable of a different type.

The system support a number of types:

- Built in primitives: `boolean`, `string`, `number` and `undefined`.
- Function types.
- Records, in practice JavaScript objects.
- The special dynamic type, annotated `?`.

Custom types may be defined with type definitions. In the implementation, it is only possible to define new record types. Record types are what statically describes a JavaScript object.

4.1 Overview of the Type Checker

The type checker works in two passes. In the first pass it parses all type definitions and annotations. Definitions are saved to the environment after parsing. Annotations, written in single line comments above statements, are parsed and then the static type is attached to the relevant node.

Since annotations and definitions are written as comments, annotated code still appears as regular JavaScript code and can be run without an extra conversion step to remove annotations and make the code runnable. They start with *::* to differentiate them from regular comments.

In the second pass, the tree is traversed again. This time with the attached static types to check the consistency of assignments, function calls, return statements, binary operators etc.

All type errors found are handled by throwing an exception, with a message detailing the kind of error and a line number. This simplified implementation was considered good enough, though it means that only one error at a time can be displayed.

4.2 The Type Checker's State

The environment or state is a data structure used by the type checker to keep track of type definitions and variable declarations. It is implemented as a stack of dictionaries, with each dictionary holding the types of variable in the current scope. When entering or leaving a lexical block new dictionaries are pushed or popped from the stack. When an identifier is encountered in the code a lookup will be performed in the environment to find its static type, starting in the local scope and checking each outer scope in turn. If none is found the dynamic type is returned instead.

The environment also holds any user-created type definitions.

4.3 Statements

The main part of the type checker program is a fold over all the nodes of the abstract syntax tree. It uses a function that test correctness given an environment and the current node produces a new environment, checking for type consistency for each statement and expression.

4.4 Expressions

Expressions, unlike statements, evaluate to a result and therefore have result types. This means that the type checking function for expressions needs to return a type in addition to the updated environment. Every part of an expression is recursively evaluated in turn to produce the resulting type.

The type system provides extra type safety by disallowing some of the implicit type conversions done by JavaScript, which is described in subsection 2.3 *The JavaScript Type System*. For example, the two sides of binary expressions have to be consistent. Consequently, something like `1 + '1'` will produce an error instead of implicitly converting the number to a string.

4.5 Variables

When declaring a variable a type annotation may be added, as exemplified:

```
//:: number
let x = 1
```

This will add the variable to the environment, associating the identifier x with the type *number* in the current scope. When assigning a value to this variable the checker will disallow values of the wrong type.

There are three kinds of variables in Javascript, *var*, *let* and *const*. Both let and const are lexically scoped, like in most languages. However, var is function scoped, meaning that a var variable defined inside an if statement will actually be hoisted to the top of the function in run-time, making it defined even outside the if block.

4.6 Functions

The type system also handles function type annotations written as

$$\texttt{T1,T2,...,Tn => R}$$

where

$$\texttt{T1,T2,...,Tn}$$

are the types of the arguments and `R` is the result type.

The arrow in the notation was chosen to mimic the way JavaScript "arrow functions" are declared. The notation is also inspired by type annotations in Haskell. The following example shows how a function that takes a number as argument and returns a number is annotated.

```
//:: number => number
let f = n => n
```

When checking a function definition, a new scope is added to the environment with mappings for the parameters. The statements and expressions are checked in turn, popping the scope as the function returns.

When applying a function, the name is looked up in the environment. After that the types of arguments are checked for consistency with the corresponding parameters on the functions type. The return type of the functions static type is then treated as the type of the function call expression. The example below shows a valid and an invalid call to the function *f*.

```
//:: number => number
let f = n => n + 1

f(1) // Fine, 1 is a number
f('foo') // Error, 'foo' is a string, not a number
```

Pure Functions

To annotate a function as pure `->` is used instead of `=>`, as examplified below.

```
//:: number -> number
let f = a => a + 1
```

The type system guarantees that a pure function is referentially transparent. It checks that a pure function

- Does not mutate its arguments, as exemplified in Listing 3.
- Does not mutate any values or update any variables outside its local scope, as shown in Listing 4.
- Does not depend on any mutable variables outside its scope. As shown in Listing 5 this would break referential transparency.
- Does not call any impure functions.

These rules ensures that a pure function will always be safe and reliable wherever its used.

It will also make sure that the function does not depend on any mutable variables in its environment. This includes global variables, but also mutable variables in its closure.

As mentioned above, pure functions can not contain calls to impure functions. However, even though there are no purity guarantees about dynamic functions,

the checker will still consider them pure. This is in keeping with the rest of the type system, that lets the dynamic parts of the code get away with anything.

```
/*:: type A { p: number }*/
//:: A, number -> A
let f = (a, n) => {
  a.p = n // Error
  return a
}
let a = { p: 1 }
f(a, 2)
```

Listing 3: The arguments in the pure function `f` may not be modified, as such this code will raise a type error.

```
//:: number -> number
let f = n => {
  a = 'foo' // Error
  return n + 1
}
f(1)
```

Listing 4: Non-local variables may not be modified in a pure function. Notice how `a` is not declared within the function scope of `f`

```
const f = () => {
    let x = 1
    //:: number -> number
    const g = n => n + x
    // g(1) == 2
    x = 4
    // g(1) == 5
}
```

Listing 5: Captured value

4.7 Type Definitions

It is possible to define custom record types. This is the way we type JavaScript objects. This may be seen in the example below. Notice that they are written in a block comment instead of a single line comment.

```
/*:: type A { x: number }*/
```

These types are parsed in the first pass, together with the type annotations. The resulting type objects are then added to the starting environment, allowing them to be looked up during checking.

As seen in the example below, the type checker will ensure that only members that are part of the type will be accessed.

```
/*:: type A { x: number; y: number }*/

//:: A
let a = { x: 1, y: 2}
a.z // Error!
```

Consistency Checking

The defining feature of gradual type systems is the ability to mix static and dynamic typing. This is done with the special dynamic type, written as *?* or *dynamic*. Variables annotated as dynamic will ignore the regular type checking constraints, allowing them to hold values of any type similar to variables in a regular JavaScript program. Shown here:

```
//:: ?
let x = 1
x = 'foo'
```

The type checker compares types using the consistency relation, as described in subsection 2.1 *Gradual Typing*. If either of the two types compared is dynamic, the relation is always true. Since all unannotated code is considered dynamic, this allows for mixing typed and untyped sections of code. In the below example, the static type of the value returned by the call to f will be number.

```
//:: ? => number
const f = d => d + 1

f('foo') // OK
```

5 Discussion: Reflection, Compromises and Alternative Routes

This section will discuss some of the design decisions made, as well as present some possible future extensions to the type system.

5.1 Dividing Impurity into Properties

During our discussion we considered categorizing functions on the basis of impurity properties that we found interesting.

Unpredictable output specifies that the result depends on a non-constant value, other than the input. For example, reading from file system.

For some functions, like those using system entropy consuming random number generation, a decision needs to be made whether they would be seen as only depending on, or actually mutating the global state.

Argument mutation describes a function that may mutate one or more of the arguments. For simplicity this was to be defined per function, but there are functions which would benefit from having per argument immutability. For example: one mutable log buffer being passed around.

Effectful includes non-local mutation and side effects.

Not all combinations of these properties are interesting, especially since argument mutation is a subset of effectful. So a decision of which which combinations would be exposed to the type language needs to be made. This is similar to what was done by Gifford and Lucassen [4].

5.2 TDD in Relation to Type Checking

In recent years the Test Driven Development methodology has become enormously popular in the software community. In short, it is a philosophy and set of practices that emphasize extensive, granular unit testing during development, often writing tests before writing functional code and aiming for 100% (or at least close) test coverage.

It's no accident that the modern iteration of TDD was invented by programmers working in dynamically type languages. While TDD is certainly useful regardless of language, as a way of raising confidence in the code, in a dynamically typed language it is absolutely essential. Making large refactoring efforts without static types can quickly turn into a nightmare, unless there are extensive unit tests in place.

In the end, static typing and automated testing are complementary methods of achieving correct programs. Type checking aims to prove properties of the program, making run-time type errors impossible. At the same time not everything can be expressed as types, and logic errors are still possible. Testing takes the opposite approach, trying to find errors by giving examples. It can find logical errors, but since testing every possible combination of inputs is impossible

there may be untested cases even with many tests, and there is always the risk of brittle tests that must be changed when refactoring causing extra work and reduced confidence in the tests.

5.3 Possible Extensions

Due to time constraints and the nature of the process the tool and the type system lacks some features which range from being optional to absolutely vital for a type checker. For the same reasons some JavaScript constructs are not supported which might affect adoption rate of already existing code bases.

Standard Library

In its current form the type checker only knows about a subset of standard library functions. Defining the built in methods for strings, arrays etc would go a long way towards making the tool more useful. This extension should be relatively easy to implement, though naturally require significant time investment to make it complete. All that would be needed are standard type definitions that are always considered by the checker.

Doing this would prevent type errors such as taking the substring of a number. It would also let us keep track of mutating methods, such as `Array.sort`.

Arrays

Another feature missing from the current type checker is handling of arrays. To do this feature justice support for parametric polymorphism, i.e. type variables, would be needed. This is because the functions written must behave the same on all arrays, regardless of the type of the elements inside. For example, a function `add` that adds an element to the end of an array and then returns it might have type `T[], T => T[]` where `T` is a type variable.

Typing arrays in this way would allow for restriction of the type of elements in an array where in regular JavaScript, a single array may contain elements of any number of types.

Parametric Polymorphism

In addition to supporting type variables in the special case of arrays, it would be helpful to support other kinds of parametric polymorphism, sometimes called generics.

Generic type definitions would allow for the creation of data structures, such as trees, without caring about the type of the elements.

Generically typed functions would also be very useful. Consider the filter function that, given a list and a predicate function, returns a list filtered with the predicate. Its type could then be `T[], (T -> boolean) -> T[]`.

Type Checking Across Files

Being able to share types between different files and allowing the type checker to access these non-local types would allow for more thorough type checking and a higher degree of type safety. Currently the type checker can only check files independently.

Complete Pureness Checking

While the type checker currently prohibits well-typed pure functions from modifying external state, no guarantees of predictable output are currently made. This is because pure functions can still depend on mutable variables outside of their local scopes.

Subtyping

Another feature that would be useful is support for subtyping. It would be interesting to explore the interaction between subtyping and consistency, since they would be independent and complementary relations.

Nominal subtyping means that subtypes are declared at type definition. This could be implemented as inheritance, for example `type Rectangle : Shape`. Here the type checker will know that a Rectangle can do at least as much as a Shape, and so any function that accepts a Shape should also be able to handle a Rectangle.

Structural subtyping means that if an object contains at least the properties and methods of a type, then it is a subtype of that type. This would be useful when creating object literals. If we have the type A `type A { x: number}` then any function accepting an A should also accept an object literal `{ x: 1}` that has the exact same structure as A.

Type Safe Recursion

In the implementation, calls to a function within that same function is treated as dynamic even if the function is statically typed. This is something that the type system should detect and handle but as of now, that is not the case.

JavaScript Constructs

Not all JavaScript-syntax is supported. The type checker will throw an error on unsupported constructs. This limits the usability of the system.

Union Types and Type Refinement

In JavaScript the way to achieve function overloading is to check the type of an argument at run-time with the `typeof` function and then branch for the

different desired functionality. Ideally the type checker would catch that inside the block the variable actually has a stricter type and make the check statically at the function call instead of inside the function at run-time. This is already possible with something like this:

```
//:: ? => ?
const f = d => {
  if(typeof d === 'number') {
    //:: number
    const n = d
    //...
  }
  if(typeof d === 'string') {
    //:: string
    const s = d
    //...
  }
}
```

However this requires an extra variable. A different way could be to introduce union types and an is keyword to the type system, letting us do something like this:

```
//:: number | string => ?
const f = d => {
  //:: d is number
  if(typeof d === 'number') {
    //...
  }
  //:: d is string
  if(typeof d === 'string') {
    //...
  }
  //this last option is invalid
  //:: d is boolean
  if(typeof d === 'boolean') {
    //...
  }
}
```

A third way is to have the type checker recognize typeof in the condition of an if statement, letting it do the above check without annotations on the if statements. A limitation with this would be that it would only work for primitives and not for our custom types (they would all be of type "object"). A solution to this problem could be to allow functions to be annotated as implementations of typeof for a certain type, which is then recognized in the if type refinement. Something like this:

```
/*:: type A { x: number } */

//:: typeof A -> boolean
const typeofA = a => a && typeof a.x === 'number'
```

```
const f = d => {
    if(typeofA(d)){
       // d must be an A
    }
}
```

Since typeofA is declared with a typeof modifier, the type checker would know that it can be used in an if condition.

In practice these methods complement each other, one might want to implement all of them.

Immutable Records

One extension that was discussed was to support immutable record types. This feature would be relatively rare amongst JavaScript type extensions, and would work well with pure functions already implemented.

The idea is to have a modifier for record types, that would mark it as immutable. Any code trying to mutate an object of an immutable type would then throw an error. This functionality should be relatively simple to implement, and work similarly to the way pure functions are not allowed to mutate arguments and variables in their closures. Like here:

```
/*:: type A { x: number } */

//:: immutable A
const a = { x: 1 }
a.x = 2 // Error!
```

One issue here is whether to make immutability the default for records, and add a mutable modifier instead. In general, immutable code is preferable, so making it the default could encourage better style.

At the same time, mutable code is idiomatic JavaScript, which might make the transition harder. But the one of the advantages of gradual typing is that the old code, without annotations, would work fine regardless.

6 Conclusion

Dynamically typed languages are used more and more and in larger and larger projects. Maintaining the code in these large code bases could prove to be problematic when all code is written dynamically and no compile time guarantees can be made. With gradual typing, these projects can be converted into statically typed code bit by bit without affecting behaviour, complexity or efficiency. Furthermore, code that for one reason or another need to be kept dynamically typed still can; as gradual typing allows for seamless, implicit conversion between both styles. One often overlooked subject in gradual typing is the concept of purity and mutability which when checked for can give the programmer even more oversight over the project in terms of unwanted side effects. With the help of projects like this and the research discussed in it, gradual typing can hopefully find a place in future software development and help turn the tides into purer Javascript.

References

[1] Microsoft. (2017-05-02). C# programming guide, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection/ (Accessed 2019-05-31).

[2] J. G. Siek. (2014-03). What is gradual typing, [Online]. Available: https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/ (Accessed 2019-02-15).

[3] P. Wadler and R. Findler, "Well-typed programs can't be blamed", 2009-03, pp. 1–16. DOI: 10.1007/978-3-642-00590-9_1.

[4] D. K. Gifford and J. M. Lucassen, "Integrating functional and imperative programming", in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86, Cambridge, Massachusetts, USA: ACM, 1986, pp. 28–38, ISBN: 0-89791-200-4. DOI: 10.1145/319838.319848. [Online]. Available: http://doi.acm.org/10.1145/319838.319848.

[5] A. Hidayat. (2012). Ecmascript parsing infrastructure for multipurpose analysis, [Online]. Available: http://esprima.org (Accessed 2019-04-18).

[6] Z. Carter. (2009). Jison, [Online]. Available: http://zaa.ch/jison/docs/ (Accessed 2019-04-18).