Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

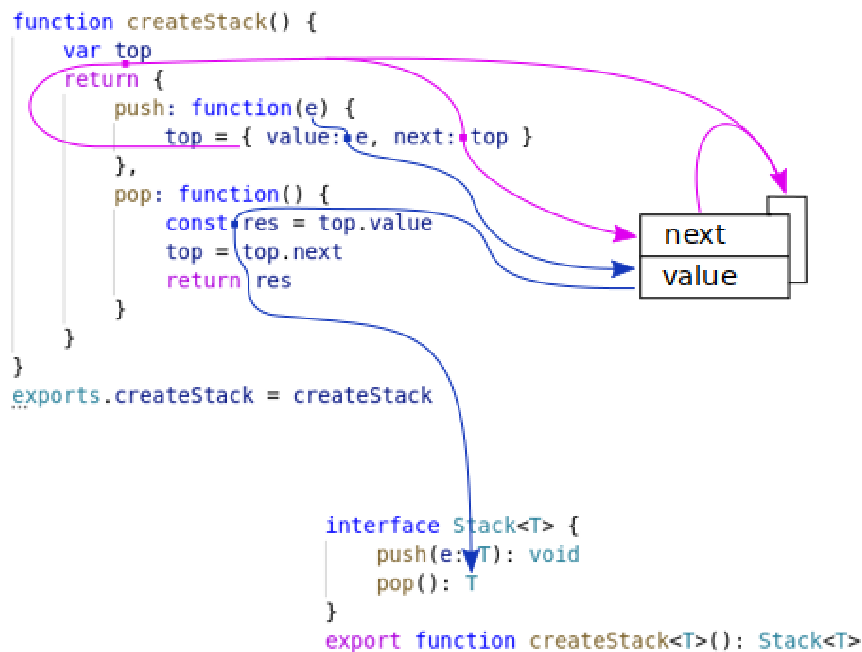
Lehrstuhl Prof. Dr.-Ing. Snelting

Data Flow Based Type Inference for JavaScript

Masterarbeit von

Tobias Kahlert

an der Fakultät für Informatik



Erstgutachter: Prof. Dr.-Ing. Gregor Snelting

Zweitgutachter: Prof. Dr. rer. nat. Bernhard Beckert

Betreuende Mitarbeiter: M.Sc. Sebastian Graf

Bearbeitungszeit: 19. Februar 2018 – 3. August 2018

Zusammenfassung

TypeScript ist eine beliebte Erweiterung für JavaScript, die es erlaubt auch typisiert zu programmieren. Funktionen und Objekte aus externen JavaScript Modulen können weiterhin eingebunden werden, erhalten aber keine Unterstützung durch TypeScripts Typprüfung.

Um diese Unterstützung nachzureichen, können eigene Typdefinitionen geschrieben werden, die die API der jeweiligen Modulen widerspiegeln. Gerade wegen der Existenz der rund 5.000 handgeschriebenen Typdefinitionen ist TypeScript sehr beliebt. Diese Zahl verblasst jedoch im Angesicht einer Gesamtzahl von ~670.000 npm-Modulen, die keine Typdefinitionen besitzen.

In dieser Arbeit präsentieren wir Inferium. Mit diesem Werkzeug lassen sich automatisch Typdefinitionen aus JavaScript Modulen erzeugen. Unsere Ergebnisse zeigen, dass Funktionstypen, Objekte, und sogar generische Parameter allein durch die Analyse von JavaScript Code inferiert werden können. Zusätzlich kann Inferium bereits vorhandene Typdefinitionen einbinden, um die Analyseergebnisse zu verbessern.

Abstract

TypeScript is a popular extension for JavaScript, that allows statically typed programming. Functions and objects from external JavaScript modules can be imported, but do not get any support from TypeScript's type checker.

To enable type checking support, type definitions that mirror the modules' APIs can be written by hand. TypeScript is popular especially because of the existence of around 5,000 handwritten type definitions. This number, however, is vanishingly small when compared to the ~670,000 existing npm modules that do not have type definitions.

To automate this manual task, we present Inferium. This tool can automatically generate TypeScript type definitions from JavaScript modules. Our results show that function types, objects, and even generic parameters can be inferred using only static analysis on JavaScript code. Additionally, Inferium can use existing type definitions to improve the results of the analysis.

Contents

1 Introduction **5**
1.1 Contributions . . . 6

2 Background **7**
2.1 JavaScript . . . 7
2.1.1 JavaScript's Type System . . . 7
2.1.2 Inheritance with prototypes . . . 8
2.1.3 Coercion . . . 11
2.2 Node.js . . . 11
2.3 TypeScript . . . 12
2.3.1 Union types . . . 13
2.3.2 Basic types . . . 13
2.3.3 Interfaces & Classes . . . 14
2.3.4 Generics . . . 15
2.3.5 Type Definitions . . . 16

3 Design **17**
3.1 Overview . . . 17
3.2 Graph representation . . . 18
3.3 Allocation-site abstraction . . . 22
3.4 Strong vs weak update . . . 23
3.5 Recency abstraction . . . 23
3.6 Assignment-site abstraction . . . 24
3.7 Probes . . . 26
3.8 Lattice . . . 27
3.8.1 A join for *State* . . . 31
3.8.2 Normalizing values . . . 31
3.8.3 Summarizing most recent objects . . . 32
3.8.4 Lattice members to TypeScript types . . . 32
3.9 Probe constraints . . . 34
3.10 Predefined TypeScript types . . . 34
3.11 Transfer functions . . . 35
3.12 Assignment site filtering . . . 36
3.13 Function calls . . . 38
3.14 Analyzing packages . . . 39
3.14.1 Termination . . . 40
3.15 Generating type definitions . . . 41

4 Implementation **45**
4.1 Heap 45

5 Evaluation **49**

6 Related Work **53**

7 Conclusion **55**
7.1 Future Work 55

1 Introduction

JavaScript is one of the most used programming language today [1]. Being the standard scripting language in the web environment, this is not likely to change any time soon either. The opposite is the case: Since the introduction of Node.js, which enables the development of desktop and server applications in JavaScript, its popularity is rising more than ever.

Because of its history and its intended use as easy-to-learn programming language for small UI scripts, JavaScript lacks features other programming languages have to support the programmer in keeping large code bases maintainable.

To counter this shortcoming, different secondary programming languages have since been developed, which offer additional programming features and often their own ecosystem, but in the end compile their code into JavaScript.

The most popular of these languages is TypeScript. As the name indicates, its main feature is the introduction of an optional type system, which not only permits the TypeScript compiler to find type errors by static type checking, but also enables editors and IDEs to support functions like code completion and refactorings.

TypeScript can use external JavaScript code, but the compiler will not be able to provide type checking or editor support for it without additional type definitions that are given to the compiler via a *type definition file*. Those files only declare the interface of a JavaScript module and do not contain any implementation themselves.

Type definitions have to be written by hand, though a public repository exists, that hosts type definitions for nearly 5,000 packages[1].

These already typed packages, however, stand against more than 670,000 packages that are currently registered on npm [2], with an average submission rate of around 400 packages that are published each day.

The manual creation of a type definition file is tedious and often error prone, as APIs are often undocumented and supporting tools are limited. dts-gen [3], for example, is a tool that generates simple type definitions for JavaScript code by executing it and examining the resulting object. This provides a good starting point for a type definition, but parameter and return types of functions are not analyzed and therefore not annotated with helpful types.

The aim of this work is to create a tool that generates type definitions that are helpful to the programmer.

The usual approach for type inference is the gathering of type constraints based on the abstract syntax tree of a given code. The patterns and idioms that are used in

[1] `https://github.com/DefinitelyTyped/DefinitelyTyped/`

JavaScript code make a purely constraint based inference unviable, because variables are often created and updated in different contexts, and even have different types at different locations.

To our knowledge, there is only one other tool that tries to generate TypeScript type definitions by analyzing JavaScript code: TSINFER [4]. It's flow-insensitive data flow analysis is a definite improvement over dts-gen, but it lacks the precision, to find more sophisticated types. It also does not take existing type definitions into account and can not process Node.js packages.

For this work, we choose a similar path as TAJS [5], an analysis tool to statically find errors in JavaScript code. It performs a flow-sensitive data flow analysis, with the ability to precisely handle JavaScript objects. Unfortunately, it is only able to perform a whole-program analysis and lacks the ability to analyze uncalled functions. This, however, is vital for our purpose of annotating parameters and return values of functions.

1.1 Contributions

We present Inferium, a tool that generates type definitions for Node.js packages. Our multi-step process loads existing type definitions, performs an extensive data flow analysis on the target package's code, and generates TypeScript type definitions.

After giving a broad overview over JavaScript and TypeScript in chapter 2, in chapter 3 we discuss the concepts our analysis uses to gather precise information about variables and properties of objects.

In section 3.2 we give an overview of the internal graph representation of Inferium's anlysis.

We describe the *allocation-site abstraction* (3.3) and *recency abstraction* (3.5) to model object allocations and introduce ***assignment-site abstraction*** (3.6) to achieve partial path-sensitivity for our analysis.

In section 3.7 and 3.9, we introduce ***probes***, elements we use to gather information about parameters in functions and from which we later infer TypeScript types and even generics.

Afterwards, we define the exact *lattice* for our analysis (3.8) and bring an example for a node's state transformation function (3.11).

In section 3.12, we describe how assignment-site abstraction can be used to achieve path-sensitivity.

Section 3.14 describes how our analysis handles whole Node.js packages and argues about the termination of Inferium's analysis.

How types are extracted from the data-flow analysis is described in sections 3.8.4 and 3.15.

In chapter 4 we discuss our approach on implementing the heap for our analysis, before we present the results of our work in chapter 5, related works in chapter 6 and our conclusion in chapter 7.

2 Background

2.1 JavaScript

JavaScript was created by Brendan Eich for the Netscape browser in 1995 as an easy-to-learn but powerful programming language. Its light syntax and dynamic nature enabled fast development for the websites of the rapidly expanding World Wide Web. Only available in Netscape, other browser manufacturers had to follow suit and implement their own versions of JavaScript. Microsoft, for example, released JScript for the Internet Explorer in 1996. Slight differences in implementation and available features made scripts incompatible between the browsers. In 1997 the language was eventually standardized under the name of ECMAScript. Since then, every other year a new version of ECMAScript is released, adding new features.

The three paradigms JavaScript uses to become lightweight but stay powerful at the same time are the lack of a static type system, coercion, and object orientation that models inheritance via prototypes. We will give a short overview over these features before discussing the way TypeScript tries to capture them in its own type system.

2.1.1 JavaScript's Type System

As mentioned before, JavaScript does not have a static type system. Rather, types are handled exclusively at runtime and are independent of variables. In the following code, for example, `x` is `0` and therefore of type number, the next line assigns a string to `y`. In the third line, however, `y` is assigned to `x`. Before that assignment `x` was of type number, while after that assignment `x` is of type string.

```
var x = 0;
var y = "I'm a string";
x = y;
```

A static type system would normally forbid such an assignment; in JavaScript this is totally fine.

JavaScript differentiates between two kind of types: primitives and objects. Primitives are handled as values that are copied when assigned, while objects are handled with references. Assigning objects just copies the references that still point to the same object.

undefined The undefined-type has only one member, namely undefined. Its the result when reading unwritten variables or properties, and the return value of functions that do not have an explicit `return` statement.

boolean The type containing `true` and `false`.

string The type of all string literals.

number The type containing all numbers. There are also the three special numbers ∞, $-\infty$, and `NaN`.

The type of a variable can be acquired at runtime by using the `typeof`-expression. `typeof true`, for example, results in the string `"boolean"`.

Behind the object type certain special objects called exotic objects are hidden. They are for all intents and purposes ordinary objects (including writable properties and a prototype chain), but their behavior differs in certain situations. Functions for example are the only objects that are callable (meaning they can be used in a call expression like `func(arg1, arg2)`). All non-function-values, in contrast, will throw exceptions when used in calls. They are also special in the way that `typeof` will evaluate to `"function"` for them. All other objects will result in `"object"`, including the `null`-value.

```
// functions are first class citizens in JavaScript
// function expression
var f = function func1() {
    return "I'm a function"
}

// function declaration (func2 is hoisted and can be called
//                        even before the declaration)
function func2() {
    return "I'm a second function"
}

typeof f === "function"
typeof func1 == "undefined"
typeof func2 == "function"
```

2.1.2 Inheritance with prototypes

Objects also hold a special internal property called prototype. If the programmer tries to read a property that does not exist on the targeted object, the object referenced as prototype will be searched. This behavior will be repeated until either the property is found or the prototype is `null`, which – by definition – has no prototype. The prototype mechanism is used to mimic object oriented inheritance, but is, in fact, more powerful. For example, prototypes can be changed for existing objects at runtime.

Objects can be created in two ways: Either by using an object expression, where the properties are directly written into the expression, or by using a constructor, where the constructor must initialize the properties. Any function can be a constructor as the following code shows.

```
function Point(x, y) {
    this.x = x
    this.y = y
}

Point.prototype.distSq = function() {
    return this.x * this.x + this.y * this.y
}

// Object created with constructor
var p = new Point(10, 10)

// p.distSq() == 200
// p.x == o.x == 10

// Object created with object expression
var o = { x: 10, y: 10 }
// o.distSq is undefined here
```

For the function `Point`, here used as constructor with the expression `new Point (10, 10)`, JavaScript will create a new empty object `p`, set its prototype to `Point .prototype`, and use it as the `this` object for the call to `Point` alongside the given arguments. Through the `this` expression the properties `x` and `y` of the object are initialized. After the call, the newly created object is returned as the result of the construction expression. A function, called `distSq`, is added to `Point`'s prototype object. When it is called on the new object, JavaScript searches the object itself. Because it only holds the properties `x` and `y`, its prototype is examined next. There the needed property exists and is called with `p` as the `this` object.

Note that most objects will have a common prototype as the end of their prototype chain. In many other languages like Java or C# this common object is called *Object*. In JavaScript `Object` exists as well, though it is not the actual common object but its constructor. `Object.prototype`, called `{}`, is the top object of most prototype chains.

The diagram in figure 2.1 shows the objects' relationships of the code above.

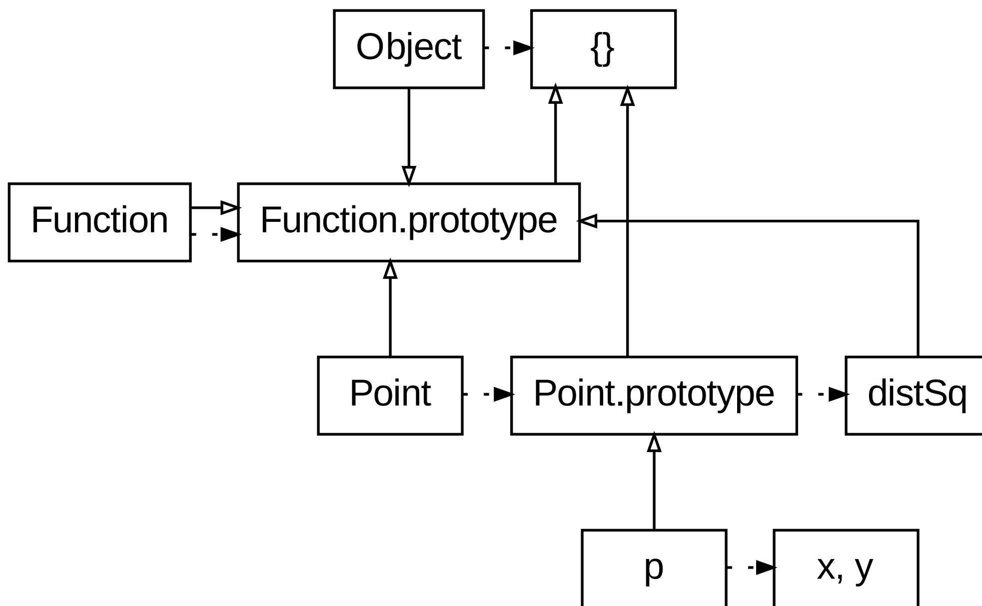


Figure 2.1: Inheritance via prototypes.
Dashed lines indicate properties, while normal lines show prototype relations. p is the newly created object that has the properties x and y. Because it was created using the Point constructor, its prototype field points to Point.prototype, which in turn has the property distSq. All functions have the prototype Function.prototype, including Function itself.

2.1.3 Coercion

The last important concept of the JavaScript type system is coercion. To handle the different types more easily, JavaScript will convert them dependent on the context they are used in. A `string` that contains a number (like `"-88"`) will be converted automatically into a number primitive when used in a calculation. This conversion is called coercion and frees the programmer from manually converting types, but the rules are also complex and can lead to confusion. Every operator has basically its own set of rules how to coerce its arguments.

The == operator itself does conversion on its arguments, which results in some curious results. The following expressions are all true.

```
!"true" == !"false"
[] == ""
8 > null
['x'] == 'x'
```

To also check for type equality the === operator exists. It first checks if both operands are of the same type and compares them afterwards.

When used in boolean contexts (in conditions of conditionals or loops), types are also coerced. Only the following values will be coerced to false: `false`, `null`, `undefined`, `0`, `NaN`, `''`, `""`. They are therefore called *falsy*. All other values are called *truthy*.

The concept of coercion is important when talking about the correct type of operators. Is it correct to say the * operator only takes numbers, when it might also take two strings containing numbers, coerce them, and multiply them correctly?

2.2 Node.js

Until the release of Node.js, JavaScript was mainly used to control websites and was running mostly in web browsers. Server-sided applications, with which JavaScript front-ends often communicate, were written in other languages. Node.js changed that by taking V8 – Google Chrome's JavaScript engine – and equipping it with built-in functions and additional libraries to enable programmers to write server-side applications.

Additionally, the Node.js team released a package manager, called npm, which enabled developers to encapsulate their code into packages, publish them into a centralized database, and in turn use others' modules within their own projects.

As JavaScript did not have its own module system (in the browser environment, different script files and libraries were imported by referencing them in the HTML code), Node.js introduced its own. Every file is itself a module. When a module is loaded, the containing JavaScript code is executed. Other modules can then be imported by using the provided `require` function and local definitions can be exported by using an `exports` object, which is also implicitly available and different for every module.

lib.js

```
// import global module
var assert
        = require("assert")

// export function hello
exports.hello
  = function(name) {
    assert(name !== "")
    return "Hello " + name
  }
```

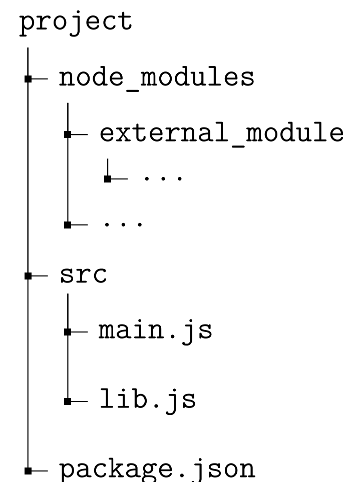
main.js

```
// import local module
var m = require("./lib")

// calling imported function
console.log(m.hello("you!"))
```

Local modules can be imported by using a relative path. Built-in modules like the above mentioned *assert* module will be imported by name. Installed packages are downloaded and put in the `node_modules` directory by npm. Node.js uses a simple algorithm to find the named modules.

If `main.js`, as shown right, want's to import module `external_module`, Node.js will cascadingly search its parent paths for the `node_modules` directory. In that directory it will search the named module and – when found – import it.



2.3 TypeScript

TypeScript was released in 2012 by Microsoft to make the development of JavaScript software more manageable. Its main goal is to enable static type checking by introducing a static type system, while staying close to JavaScript, syntax wise. The TypeScript compiler will therefore take TypeScript code, check the code for type errors, and output JavaScript code.

A type of a variable is determined at its definition by either type inference or explicit annotation. In the following TypeScript code, the parameter `left` and `right` are explicitly typed by stating their types after a ":". The type of the variable `result` is not explicitly specified, but rather infered implicitly by looking at the type of the initialization expression, which is number in this case. The return type of the function is infered similarly.

```
function subtract(left: number, right: string) {
  var result = left - right
  return result
}
```

In the code above `right` is annotated to be a string, which is not allowed as the right operand of the subtract operator. Consequently, the compiler will generate the following error message:

```
The right-hand side of an arithmetic operation must be of
   type 'any', 'number' or an enum type.
```

Despite of the error, TypeScript is able to generate the corresponding JavaScript code, which looks quite similar and has only the type annotations removed.

```
function subtract(left, right) {
    var result = left - right;
    return result;
}
```

2.3.1 Union types

Before we have a closer look at the basic types, we have to examine union types. As shown in the first JavaScript example, variables can have different types in different contexts. To be able to type variables that may be of different types, TypeScript introduces union types, which combine the set of possible values of different types.

```
var x: number | string  = 0;
var y: string           = "I'm a string";

// no type error here, because x can be a number as well as a
   string
x = y;
```

2.3.2 Basic types

TypeScript's builtin types are modeled after JavaScript's runtime types to stay as close to the target language as possible.

In addition to the counterparts of JavaScript's primitives (undefined, boolean, string, symbol), TypeScript comes with a few additional types

any The any type can be assigned to every other type and every other type can be assigned to any. Essentially, it disables type checking for a variable annotated with any.

never This type indicates that an assignment of it does never actually happens at runtime. It can be used as the result of functions that never return (because of endless loops or because they always throw an exception).

Note, that for any type X the type never | X is equal to X.

null, undefined Like undefined, null can be `null` as only possible value. By default every variable can implicitly be `undefined` or `null`, though with version 2.0,

the compiler option *–strictNullChecks* was introduced to TypeScript, removing this implicitness. If a variable can be `undefined` or `null` it has to be explicitly stated. In our analysis both types will be handled independently of one another.

void void is used as return type of functions that don't return anything. Only undefined and null are assignable to it.

object The object type contains all objects. Note that object differs from Object, which denotes the class of JavaScript's {} object.

In some situations it is handy to be more specific than using a primitive type. For booleans, numbers, and strings TypeScript introduces literal types, which can only hold one specific value.

```
var greeting: "Hello" | "Welcome" = "Hello"
var everything: 42                = 42

greeting = "Welcome" // ok
everything = 0       // error
```

2.3.3 Interfaces & Classes

Objects can be typed by writing their properties and their respective types in between curly brackets.

```
var o: { x: number, y: number } = { x: 10, y: 10 }
o.distSq() // Error: Property 'distSq' does not exist on type
    '{ x: number; y: number; }'.
```

Next to the type and existence of properties, they can also be annotated with additional modifiers. TypeScript supports for example `readonly` to prevent writing to properties.

To make the notation more convenient, object types can be named using interfaces and classes. Classes will additionally create constructor functions and prototype objects.

For checking the assignment of objects, TypeScript uses structural subtyping. An object type S can be assigned to an object type T if S has at least all properties of T and the types of these properties also are subtypes of their respective types in T.

```
// equivalent to { distSq: () => number }
interface Length {
  distSq(): number
}

// "implements Length" forces Point to implement the distSq
   method, but won't change the prototype.
class Point implements Length {
  readonly x: number
  readonly y: number

  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }

  distSq() {
    return this.x * this.x + this.y * this.y
  }
}

var p: Point = new Point(10, 10)
p.x = 39            // Error: x is readonly
var l: Length = p   // Ok

// Ok, because of structural subtyping
var l: { readonly x: number } = p
```

2.3.4 Generics

TypeScript supports generics for functions, interfaces, and classes. Upper bounds can be used to narrow generic types. Type parameters are always handled bivariantly even if that can later lead to type errors at runtime, which is the primary reason that TypeScript's type system is unsound.

```
// T has to be an object
interface Box<T extends object> {
  value: T
}

function boxed<T extends object>(v: T): Box<T> {
  return { value: v }
}

boxed(8) // Error: number not assignable to object
var aBox: Box<{ a: number }> = boxed({ a: 0 }) // Ok
```

```
// Ok, because of bivariance
var objBox: Box<{}> = aBox

// Ok, but aBox points to an empty object now
objBox.value = {}
```

2.3.5 Type Definitions

Code written in TypeScript can be checked by the compiler and converted into JavaScript code. Because JavaScript is the target language, TypeScript can also call functions and use objects that are defined in JavaScript code (by installed packages, for example). When the compiler performs type checking, however, it knows nothing about these functions and will report a missing definition or a type error. In fact the `require` method, that can be used to import foreign and in JavaScript written modules, has `any` as return type. This is a working but unsatisfactory compromise. The compiler will not throw any type errors, but it will not report wrong usage either. Additionally many editors are capable of providing type information about functions and objects for TypeScript code to the programmer. This is especially helpful in the case of modules that are written by other people and of which the programmer has not much knowledge himself. The `any` type is not very helpful in that case.

To still use external JavaScript code with full support for type checking, TypeScript has the ability to read so called type definition files (recognizable by the file ending `.d.ts`). These provide types for global definitions and for the `exports` object of modules, which can then be used by the compiler to offer full type checking support.

Type definitions look like normal TypeScript code but without the actual implementation of functions.

```
// Length and Point are not exported,
// and are therefore not members of the imported module type.
// They will however be available as types
export interface Length {
  distSq(): number
}

export class Point implements Length {
  readonly x: number
  readonly y: number

  constructor(x: number, y: number)

  distSq(): number
}

// hello is exported and is available on the import object
export function hello(name: string): string
```

3 Design

In this chapter we will present the design of our solution including the AST transformation into a node-based graph representation, the lattice that is used to describe the state of our abstract interpretation of the Node.js package, the transfer function that transforms the abstract state depending on the node type, and the final type generation.

3.1 Overview

The goal of our project, Inferium, is to generate type definitions for a given Node.js package.

To accomplish this we perform multiple steps to gather information about the targeted package and then start an extensive data-flow analysis to determine in which contexts which values can occur in which variables and properties.

We are only interested in the values that can occur as arguments, return values, and those which are reachable through the exports object, because those will directly be printed out in the final type definition. Objects and functions that are only internally used by Node.js modules are, nevertheless, very important, because they might influence the values of other variables that are printed later.

Inferium performs the following steps of which 5-8 are the steps of the actual data-flow analysis.

1. Download the target package and its dependencies.
2. Load type definitions for dependencies that are already typed.
3. Parse the code of the main module into an AST and transform it into a graph representation.
4. Create an analysis state that contains all predefined JavaScript objects and functions.
5. Use this state to analyze the main module using the data-flow analysis. If further modules are needed, load their code.
6. The resulting state is the state in which the client code gets control. It holds functions which could potentially be called by the client code.
7. Find these user-callable functions and analyze them. Use probes as arguments for each function to record how the arguments are used.

8. Join the state resulting from the analysis of the user functions with the user state and repeat steps 5 - 8 until the user state doesn't change anymore. Note that return values can themselves contain new user-callable functions.

9. Extract the possible parameters and return values, condense them into types, and print the TypeScript type definitions.

3.2 Graph representation

To analyze the JavaScript code, we transform the AST of the parsed code into a graph representation. Each node in the graph represents the transformation of one or more incoming execution states into a new execution state. Edges model the potential control-flow between nodes. Control-flow means that the resulting execution state of one node is given to another.

A single execution state models the full state of a JavaScript engine at a particular moment in the execution of a program. Chapter 3.8 describes the execution state in detail. Here we will give only a brief overview:

Expression Stack Some nodes produce temporary values that need to be passed to subsequent nodes. In the expression `!true`, for example, the `true` is generated by a node and has to be passed to the node that models the *not* operator, which in turn will further process the `true`. There are several ways to model this like additional data edges through which the values are passed directly and independent of the control flow edges. In our analysis, we choose a stack based approach, where the temporary values are pushed on and popped off an expression stack.

Heap All objects and their properties that exist at a given moment are represented as a partial map from object labels to object descriptors, which in turn hold mappings from property names to property descriptors. We call this the heap.

This Every state has a current this value. It is only used in the `this` expression, where it is the resulting value of the expression.

Lexical Frames The Lexical Frames are a chain of objects, which store the local variables of the current scopes. Each frame refers to a different scope, starting from the most outer scope. For example, a state for a function that is defined in the top-most scope — the module scope — has at least two lexical frames: the frame for the module-scope and the frame for its own function scope.

Each node in the graph performs a fundamental transformation of the execution state, so that a minimal set of different nodes is required to represent a JavaScript program. The following nodes exist:

merge Joins all incoming states into a new state.

cond *t_1*, *t_2* A conditional jump. Reads a value from the stack. If the value might be truthy, control flow goes to t_1. If it might be falsy, control flow goes to t_2. Note, that control flow can be passed to t_1 **and** t_2. Here we also perform a filtering of the condition (See 3.12).

throw Takes a value from the stack and gives control to the next catch target if that is not empty. The taken value is used as exception.

ret This node marks the end of a function and handles the return value.

end This node marks the end of a script and has no other purpose than to capture the incoming execution state.

push *L* Pushes the literal L onto the stack.

binary *Op* Takes two values from the stack and applies the binary operator Op. The following operators exist: `!=`, `!==`, `%`, `&`, `*`, `**`, `+`, `-`, `/`, `<`, `<<`, `<=`, `==`, `===`, `>`, `>=`, `>>`, `>>>`, `^`, `in`, `instanceof`, `|`

unary *Op* Takes a value from the stack and applies the unary operator Op. The following operators exist: `!`, `+`, `-`, `typeof`, `~`

pop *N* Erases the first N elements from the expression stack.

dup *N* Duplicates the topmost stack value N times.

dup2 Duplicates the two topmost stack values.

pushLex Creates a new object and adds it to the lexical frames.

readL *S* Reads the variable S in the current lexical environment.

writeL *S* Writes the variable S in the current lexical environment.

pushThis Pushes the *this* value of the current state onto the stack.

call *N* Takes a function object and N arguments from the stack and calls the function.

invoke *N* Takes a value, a function object, and N arguments from the stack and calls the function with the value as new this value.

allowObj Creates a new object and pushes it onto the stack.

allowArray *N* Creates a new array and initializes it with N values from the stack.

allowFunc *F* Allocates a new function object for function F.

new *F* *N* Takes a constructor function F and N arguments from the stack, creates a new object, and executes the function with the newly created object as *this* object.

readP *S* Takes an object from the stack and reads the property S.

writeP *S* Takes an object and a value from the stack and writes the value to the property S.

readDyn Takes an object and a property name from the stack and tries to read the property on the object.

writeDyn Takes an object, a property name, and a value from the stack and tries to write the value into the property of the object.

When an AST is transformed into a graph, the transformation process ensures that every node gets additional information like a *lexical environment* and a *catch target*. The lexical environment is a map from existing variable names to an offset of the lexical frames and is used to determine the exact lexical frame on which local variables are read. Consider the following function.

```
var x
var y

function test() {
  var x

  x = 42
  y = 42
}
```

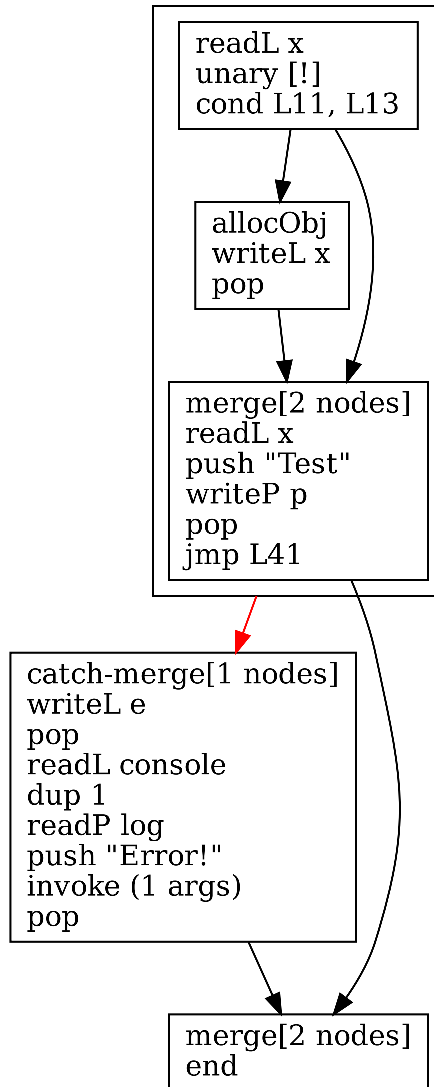
In function `test`, the outer lexical frame holds `x` any `y`, while the inner lexical frame only holds `x`. When the two assignments in `test` are analyzed, the writeL nodes need to know on which lexical frame to find `x` and `y`. This information is stored in the lexical environment, which has offset 1 for `x` and offset 0 for `y` for both writeL nodes in the above example.

The aforementioned catch target is either the merge node which starts the catch graph of the nearest enclosing try-catch statement or empty in case there is no enclosing try-catch statement.

Figure 3.1 shows a representation of the graph where all nodes that have a single predecessor or a single successor are merged, so that one box represents multiple consecutive nodes.

```
try {
  if (!x) {
    x = {};
  }
  x.p = "Test"
} catch (e) {
  console.log("Error!")
}
```

a A simple program with control flow and a try-catch-block.



b The produced graph

Figure 3.1: An example program and its graph representation. Consecutive nodes are shown in one box. The catch-merge is a normal *merge* node which gets control flow when one of the nodes in the linked box might throw an exception.

3.3 Allocation-site abstraction

As many other languages JavaScript supports local variables as well as dynamically allocated objects.

Local variables normally exist only once per function frame (ignoring closures) and are therefore easy to analyze via data-flow analysis, because operations that use those variables will know exactly on which values of the program they operate.

For operating on properties that are part of allocated objects this is not true, generally. Consider the following code.

```
var a
var b

for (var i = 0; Math.random() > 0.5; ++i) {
  var o = { index: i }
  if (Math.random() > 0.5) {
    a = o
  } else {
    b = o
  }
}

console.log(a.index)
console.log(b.index)
```

Not only do we not know to which object `a` or `b` actually point, we don't even know how many objects are created in the loop.

This is a major problem for the analysis, because it still needs to handle reads and writes to the property `index`, which can exist potentially infinitely often. We can't create a new object for the analysis every time the analysis processes the allocObj node either, because we still have to find a fixed-point of the analysis, which won't exist if we keep adding objects.

To handle this, we use the *allocation-site abstraction* [6] [7]. With that, we create a single abstract object, labeled l, for every allocObj node in the graph. l represents all objects that are possibly created at that particular allocation site.

In the above code we can now easily see that `a` as well as `b` are pointing to the same abstract object while also potentially being `undefined`, which models the possibility that they are not initialized at all.

With this technique, finding a fixed-point does not become impossible, because both variables won't change in a second round of analyzing the loop. Same is true for the property `index`, which only exists once.

3.4 Strong vs weak update

When dealing with abstract objects, where multiple potential objects are referred to by one single label per allocation site, updating properties becomes a problem, because the analysis does not know to which object a value is written. Unfortunately, this leads to imprecision when combined with followup reads like in the following code.

```
var a
while(Math.random() > 0.5) {
     a = {}
     a.prop = "test"
     console.log(a.prop)
}
```

Here it is obvious that all objects that are created inside of the loop must have a property `prop`, and that `a.prop` is always `"test"` when printed in the last line. But because `a` always refers to all potentially created objects, the write to `prop` immediately after the object allocation, has to make sure that `a.prop` can be potentially absent, resulting in `a.prop` being undefined at the time it is printed. We call this a *weak update*. Respectively, we call writes to objects where the analysis can be certain that only a single object is written to *strong update*.

3.5 Recency abstraction

Because objects are allocated independently from their initialization in JavaScript, it is important that writes to newly created objects are strong updates. In the last section, we examined exactly such a situation. The object is allocated in one line and initialized in the next line.

To enable strong updates on newly created objects we use the *recency abstraction* [8]. The idea behind the recency abstraction is to use two abstract objects per allocation site. One is the object that refers to the most recently created object at an allocation site. We label it $l^@$ and call it most recent object. The other object, which we label l^* and call summary object, models all other objects that were potentially allocated at that exact same allocation site.

In our analysis, allocObj will take the incoming state and push a label $l^@$ of the most recent object for that allocation site onto the stack. At the same time, the properties of the most recent object are joined into the summary object. To differentiate between the new reference to $l^@$ and the already existing references, all existing $l^@$ that had been in the incoming state will be changed to the summary counterpart l^*.

Note that the recency abstraction is not the solution for all situations where strong updates are possible.

```
var a
if (Math.random() > 0.5) {
  a = { prop: "hello" }
} else {
  a = { prop: "world" }
}

a.prop = "bye!"
console.log(a.prop)
```

Here, `a` could potentially point to two different most recent objects and the assignment of `"bye!"` to `a.prop` has to be weak. This leads to three possible values that could be printed in the next line: `"hello"`, `"world`, and `"bye!"`. A strong update of the property would have been desirable, because we know that `a` can only point to one of the two objects and therefore the result of `a.prop` in the log expression must be the same value which was assigned to it a line prior, regardless to which object `a` actually points.

Our analysis does not support this kind of strong update, but in the Future Work section 7.1, we describe a possible solution for this kind of problem.

3.6 Assignment-site abstraction

Every time branching is involved, our analysis has to analyze all branches, except it can prove that the condition for a branch is not met.

In the case that we have to analyze all branches, we can still learn something from the conditions of the branches. Namely, that the condition was met for the branch that we have to analyze, which is called path-sensitivity.

Consider the following code, for example.

```
var obj = {}
if (Math.random() > 0.5) {
  obj.a = "Hello World"
}

if (obj.a) {
  console.log(obj.a)
}
```

After the first if block, we know that `a` can be `"Hello World"` or **`undefined`**. When we enter the then branch of the second if statement, however, we could have learned from the condition, that `obj.a` can only be `"Hello World"` because the condition demands it to be truthy, which **`undefined`** is not.

But how can we filter the possible values of `obj.a` when the if condition in the data-flow analysis only gets handed its two actual values and every reference to their origin is lost?

A simple solution would be for the `readP` node to return a reference to `obj.a`, which is handled like a normal value, instead of the actual value of the property. This

reference could, for example, encode the base object and the name of the property.

When such a reference is then used in the context of a branching condition, we can filter the values of the property to which the reference points to only those values that would meet the condition. Because we handle these references just like normal values, this would even work over multiple writes to other variables or properties, as the next slightly modified version of the previous example demonstrates.

```
var obj = {}
if (Math.random() > 0.5) {
  obj.a = "Hello World"
}

var condition = obj.a

if (condition) {
  console.log(obj.a)
}
```

Here, `condition` holds the reference to `obj.a` instead of the two actual values. When it is then used in the branching condition we can still enact the filtering and have effectively changed both variables.

The problem with this solution is when the property a reference points to is newly assigned. This is illustrated in the following code.

```
var obj = {a: "Hello World" }

var condition = obj.a

obj.a = "Wrench"

console.log(condition)
```

When `condition` is printed, it still points to `obj.a`, which holds an entirely different value than at the time `condition` was assigned.

To solve this, we introduce the *assignment-site abstraction*. The basic idea is to use references like we discussed before, but instead of letting them simply point to object properties, we have to let them point to a property at the specific time it is read, so that the value to which the reference refers is not affected when the property itself is changed. We can easily see that multiple consecutive reads of the same property without updates in between refer to the same value. Therefore, our reference has to point to the value which a specific update wrote into a property.

To summarize, a read of a property has to return a reference to the last update of the property. When the property is changed afterwards, the reference still points to the exact same update.

For this we introduce a new label a for every site in our symbolic interpretation where a property is updated, which we call *assignment site*. Properties no longer hold

values directly but via labels to assignment sites. Like objects, values of assignment sites are stored in the heap via a partial map that assigns values to assignment sites.

The above example has two updates to `obj.a`: the initial initialization `a: "Hello World"` (a_1) and the later update `obj.a = "Wrench"` (a_2).

When `condition` is assigned, the read of `obj.a` yields a new reference that points to the site where the property was last assigned, which is a_1. Later the property is updated and holds a_2, but `condition` still points to a_1 and a_1's value is stored in the heap and can be accessed when `condition` is printed.

Because properties might get joined when two control flow branches merge, every property and every reference respectively, has to be capable of holding multiple potential assignment sites.

3.7 Probes

One key capability of our analysis is the ability to analyze functions that are never called. This is important, because Node.js packages often do not have a `main` function that executes the package's relevant code but rather export functions, which the user can call.

When an uncalled function is analyzed, the first question we have to answer is what we pass as parameters. We can not simply omit them (which is equal to passing **`undefined`** in JavaScript) because that would lead to subsequent failures in the analysis and we wouldn't learn anything about the types of the parameter, either.

Instead, we pass a hypothetical value, which we call *probe*. A probe behaves as if it could be any JavaScript value. It is truthy and falsy at the same time, for example. To distinguish between multiple probes, we assign each probe a label $p \in P$ dependent on the site P where we introduce the probe into the analysis. For example, each parameter of every function has its own probe label, so when we analyze a specific function, we know exactly which probes to pass as arguments.

When probes are used in the analysis, we add constraints to a global constraint system, which we later use to extract type information about the probe and with that about, the parameter.

```
function f(param) {
  return param - 7
}
```

In the above code for example, the probe p_1 that is created for `param` is used in a subtraction. When we reach the `binary` node, which represents the subtraction, we see that its left value contains the probe p_1. Because we need a number on the left side of a subtraction, p_1 behaves as if it were a number, while we also add the constraint that p_1 has to be a number to the global constraint system.

Not only parameters, but some nodes can also introduce new probes themselves. What is, for example, the result when we read a property *prop* on a probe p?

We are definitively interested in constraints about that result and so we create a new probe and add the constraint, that a `p.prop` must result in this new probe.

As the following example demonstrates, however, we can not simply create a new probe for every property of every probe!

```
function f(param) {
  while (Math.random() > 0.5) {
    param = param.prop
  }
  return param.prop
}
```

Given that `param` contains the probe p_1, we create a new probe p_2 when we process the property access `param.prop` and assign this new probe to `param`. The next time we analyze the loop, `param` can be p_1 and p_2. For p_1, we already know that a read of `prop` will result in p_2, but for the read on p_2 we have to create a new probe p_3. Because we create more and more probes inside of the loop, we will not be able to find a fixed-point for this loop.

We solve this by introducing a probe per node that reads a property. With this, the read on `param.prop` will always result in the probe p_2. Subsequently, we will add the constraints that a read of `prop` must result in p_2 for p_1 as well as for p_2.

Because probes are handled as if they were real values, we can even infer generic type parameters when we generate the type definitions in the final step of Inferium. The above code would result in the following type definition.

```
interface I<T> {
  prop: I<T>
}

function f<T>(param: I<T>): I<T>
```

3.8 Lattice

In the following section we describe the lattice that models the execution states. An abstract state at a node N models the possible states a JavaScript engine might have after executing the given program until node N. Our lattice is similar to the lattice of TAJS [5], which itself is similar to a lattice for constant propagation with the addition that we support assignment sites and probes. We define a lattice for values, a lattice for the heap, and finally a lattice for the whole execution state.

The $Value$ lattice supports JavaScript's primitives and objects. Because TypeScript supports string-literal types, which often appear in union types with other string-literals, our lattice can keep track of multiple different concrete strings per value.

For a graph G, we let N denote the set of nodes, L the set of object labels that correspond to allocation sites, A the set of assignment labels that correspond to assignment sites, P the set of probe labels referring to probe sites, and S the set of

possible strings. To keep track of functions and recognize recursion in the state, each function definition has its own label $f \in F \subseteq L$.

We define two different value lattices:

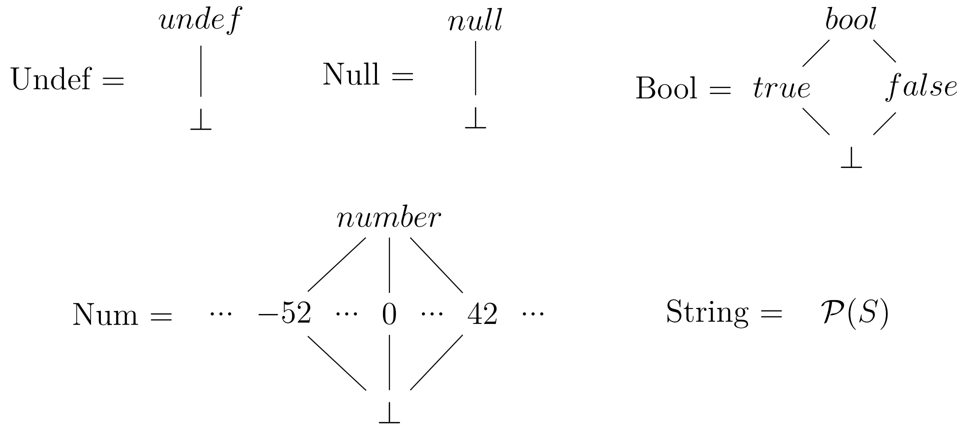
$$NormValue = Undef \times Null \times Bool \times Num \times String \times \mathcal{P}(L) \times \mathcal{P}(P)$$

$$Value = NormValue \times \mathcal{P}(Ref)$$

$$Ref = A$$

While $NormValue$ only models actual values, $Value$ can indirectly refer to actual values via references, which have to be resolved using the heap. A reference is simply an assignment site label.

The components of $Value$ and $NormValue$ are defined as follows:



In JavaScript objects are handled by references and not by value. We model this by using object labels in the $Value$ lattice, and by providing an additional partial mapping in the state lattice from object labels to object descriptors, which we call the $Heap$ lattice. Normally, each object label references its own allocation site, so that all objects that could possibly be created at an allocation site will have the same label. As we discussed before, this makes it hard to precisely handle the initialization of objects and we therefore use the *recency abstraction* [8]. For every allocation site we use two abstract objects and two labels. The label $l^@ \in L^@ \subseteq L$ refers to the most recent object and the label $l^* \in L^* \subseteq L$ to the summary object, which represents all non recent objects. Allocation nodes that create new objects will transform all most recent object labels of the incoming state lattice into summary object labels.

$$NormHeap = (L^* \hookrightarrow SumObjDesc)$$

$$MRHeap = (L^@ \hookrightarrow MRObjDesc)$$

$$Assignments = (A \hookrightarrow Value)$$

$$Heap = NormHeap \times MRHeap \times Assignments$$

The *Heap* lattice describes the heap of an execution state. It contains two partial mappings for objects – one for the most recent objects and one for the summary objects. Additionally, it contains a partial mapping that assigns values to assignment sites.

Objects are further described by a partial mapping from possible property names to properties, and two additional components: A property that is used to handle reads and writes to the object when we can not determine the concrete property name, and a set of object labels that model the possible prototypes of the object.

Most recent objects have their own descriptor, because they can also hold *references* in their properties, which the summary objects can not.

Those object labels, that describe the allocation site of a function, also have an internal binding to a *function descriptor*. This descriptor can either be the starting node of the function's graph representation, a built-in function, or an imported function for which we only know the type. More information on how this descriptor is used can be found in section 3.13.

$$MRObjDesc = (S \mapsto MRProperty) \times MRProperty \times \mathcal{P}(L)$$

$$SumObjDesc = (SProperty) \times Property \times \mathcal{P}(L)$$

For every property, we precisely capture all attributes that JavaScript has for its properties, including if they can be modified by `Object.defineProperty` ($Configurable$), are iterated over in a `for in` loop ($Enumerable$), and if they are assignable by an update expression ($Writable$).

We differ between properties for most recent objects and summary objects. While summary objects only hold normalized values (values that do not contain *references*), most recent objects use only assignment sites to model which values they are holding.

The component *Absent* also models if a property might be absent. Properties of most recent objects do not have the *Absent* component. For them we model potentially absent properties by pointing to a specific assignment site a_{absent}, called *absent site*, that always holds the value $\bot$ and is only used to indicate that a property might be absent.

$$MRProperty = \mathcal{P}(A) \times Configurable \times Enumerable \times Writeable$$

$$SumProperty = NormValue \times Configurable \times Enumerable \times Writeable \times Absent$$

The components of the property definitions are defined as follows.

$$\text{Configurable} = \begin{array}{ccc} & \top & \\ \diagup & & \diagdown \\ conf & & notConf \\ \diagdown & & \diagup \\ & \bot & \end{array} \qquad \text{Enumerable} = \begin{array}{ccc} & \top & \\ \diagup & & \diagdown \\ enum & & notEnum \\ \diagdown & & \diagup \\ & \bot & \end{array}$$

$$\text{Writable} = \begin{array}{ccc} & \top & \\ \diagup & & \diagdown \\ write & & notWrite \\ \diagdown & & \diagup \\ & \bot & \end{array} \qquad \text{Absent} = \begin{array}{c} maybeAbsent \\ | \\ \bot \end{array}$$

By default, *MRObjDesc*'s property mapping maps to $(\{a_{absent}\}, \bot, \bot, \bot)$.
Similarly, *SumObjDecs* maps to $(\bot, \bot, \bot, \bot, maybeAbsent)$ by default.
The default values are needed for the join of property mappings. When two object descriptors are joined, we do not want to loose the information, that one of their properties was not assigned. The *absent* gets removed, when a strong update is performed on a property of a most recent object. In that case the mapping will be updated to a new property that is not absent.

We can now define the state lattice which models an execution state of a JavaScript engine at a given time in the execution of a program. A state consists of a Heap, a Stack, a This value, and a scope chain, that holds the current and the outer scope objects.

$$State = Heap \times Stack \times This \times ScopeChain$$

$$Stack = Value^*$$

$$This = Value$$

$$ScopeChain = \mathcal{P}(L)^*$$

Finally the *AnalysisLattice* describes the analysis by assigning abstract States to each node. The assigned states are meant to model the state **before** the execution of the node.

$$AnalysisLattice = V \times N \rightarrow State$$

V is a set of version names that are used to differentiate between different call contexts. We use $V = (F, N)^*$, a stack of pairs of function labels and call nodes.

3.8.1 A join for *State*

An integral part of a lattice is the definition of the join operator $\sqcup$. States, for example, are joined at merge nodes, which occur at any point where control flow from different nodes must flow into the same target node.

The join for Hasse diagrams, Cartesian product, tuples and (partial) maps are defined as usual.

Because our complete lattice is made from these basic building blocks, we do not need to define our own join as it follows from the usual definitions.

3.8.2 Normalizing values

We use references to model partial path sensitivity in our analysis. References are handled like values but do not contain their actual value themselves, but point to them via assignment sites. The actual values for each assignment site are stored in the $Heap$.

In the transfer function of most nodes, we are not interested in the references but in the value they point to. Because nodes get their arguments from the stack and the *Stack* lattice is a list of $Value$, we need a function to convert $Value$ to $NormValue$, which normalizes the references to actual values. For this we introduce *normalize* .

$$normalize\colon Value \times Heap \to NormValue$$

normalize is defined as the least solution to the following equation.

$$\begin{aligned} normalize((u, nl, b, n, s, o, refs), h) \sqsupseteq & (u, nl, b, n, s, o, \bot) \\ & \sqcup \bigsqcup_{r \in refs} normalize(h_{Assignments}(r), h) \end{aligned}$$

3.8.3 Summarizing most recent objects

When a node allocates a new most recent object with label $l^@$, it also has to bring the information about of the most recent object with the same label in the incoming state into the summary object l^* of the new state. We call this *summarizing.* A most recent object descriptor is summarized with the *summarize* function, which maps the properties in the descriptor using the $summarize_{prop}$ function, which is defined as follows:

$$summarize_{prop}\colon MRProperty \times Heap \to SumProperty$$

$$summarize_{prop}((sites, c, e, w), h) = (value, c, e, w, isAbsent)$$

with

$$value = normalize_{prop}(\bigsqcup_{s \in sites} h_{Assignment}(s), h)$$

$$isAbsent = \begin{cases} maybeAbsent & \text{if } l_{absent} \in sites, \\ \bot & \text{otherwise} \end{cases}$$

3.8.4 Lattice members to TypeScript types

For the probes' constraints and eventually in the type generation phase we have to extract TypeScript types from the $NormValue$ lattice members. $toTS$ defines the transformation from lattice member to TypeScript types. These types and their join are defined by TypeScript. The transformation is simple as every lattice member has a pedant in TypeScript. Only the probes are problematic as they are additional constructs for the analysis. Nonetheless do we need them in the resulting types to be able to process them in the final step of Inferium, where we generate the type definitions. We therefore extend the TypeScript type system to contain *probe types.* Every probe type ϱ_p contains the label p of the probe it was constructed from. They behave like literal types in the lattice structure of the TypeScript type system.

$$\text{toTS}_{Undef}(x) = \begin{cases} undefined_{ts} & \text{if } x = undef, \\ never_{ts} & \text{otherwise} \end{cases}$$

$$\text{toTS}_{Null}(x) = \begin{cases} null_{ts} & \text{if } x = null, \\ never_{ts} & \text{otherwise} \end{cases}$$

$$\text{toTS}_{Bool}(x) = \begin{cases} boolean_{ts} & \text{if } x \in \{bool, true, false\}, \\ never_{ts} & \text{otherwise} \end{cases}$$

$$\text{toTS}_{Num}(x) = \begin{cases} number_{ts} & \text{if } x \in \{number\} \sqcup \mathbb{N}, \\ never_{ts} & \text{otherwise} \end{cases}$$

$$\text{toTS}_{String}(x) = \begin{cases} string_{ts} & \text{if } x = \top_{String}, \\ \bigsqcup_{s \in x} "s"_{ts} & \text{otherwise} \end{cases}$$

$$\text{toTS}_{Absent}(x) = \begin{cases} undefined_{ts} & \text{if } x = maybeAbsent, \\ never_{ts} & \text{otherwise} \end{cases}$$

Note that the properties of an object's prototype are simply joined with that of the object itself. This is to simplify the type conversion, because the prototype mechanism is more powerful than the simple inheritance that TypeScript supports.

$$\begin{aligned} toTS_{Obj}(o, h) = & toTS_{Obj}(proto, h) \sqcup \\ & \bigsqcup_{s \mapsto (p, _, _, _, a) \in m} \{s \colon toTS_{NormValue}(p, h) \sqcup toTS_{Absent}(a)\}_{ts} \\ & \text{with } (m, dyn, proto) = h_{NormHeap}(o) \end{aligned}$$

$$\begin{aligned} toTS_{NormValue}((u, nl, b, n, s, os, ps, h)) = & \\ & toTS_{Undef}(u) \\ & \sqcup toTS_{Null}(nl) \\ & \sqcup toTS_{Bool}(b) \\ & \sqcup toTS_{Num}(n) \\ & \sqcup toTS_{String}(s) \\ & \sqcup \bigsqcup_{o \in os} toTS_{Obj}(o, h) \\ & \sqcup \bigsqcup_{p \in ps} \varrho_p \end{aligned}$$

3.9 Probe constraints

To extract information about parameters, we use probes. These lattice members behave like if they could be any JavaScript value and additionally record how they are used. For this recording, nodes add constraints to a global constraint system if they encounter a probe. When we generate type definitions in the end, we can use these constraints to extract information about the type of parameters and return values. Because the constraints are global, we cannot use lattice members inside of these constraints, because they each would need a heap to be meaningful. We therefore use only TypeScript types and probe types in this constraint system. We use the following constraints.

$write(p, s, ty)$	A type ty was written to property s on p
$read(p_1, s, p_2)$	p_2 was the result of a read of s on p_1
$useAs(p, ty)$	p was used in a context where a type ty was reuqired
$call(p_1, tys, p_2)$	p_2 was the result of a call to p_1 with arguments tys
$new(p_1, tys, p_2)$	p_1 was constructed with new and arguments tys returning p_2

3.10 Predefined TypeScript types

Inferium will load existing TypeScript definitions for dependencies of target packages. For that, we first have to download their existing type definitions. If a package is called *packagename*, for example, the packages' type definition are in a package called *@types/packagename*. This package might not exist, but we try to install it nevertheless and ignore any error that may occur during the installation.

After all type definitions of dependencies have been downloaded, Inferium generates a TypeScript file that imports all dependencies and assigns them to variables. Then it uses the *TypeScript compile*, to analyze this file and gathers type information about the imported dependencies as well as about the *global object*.

This information is compressed into a json file and handed to the core analysis, which in turn loads the types from the file.

When we create the initial heap state, the type of the global object is traversed and built into real lattice objects.

With this additional knowledge, we wanted to improve the analysis' results and also improve performance by avoiding the need to analyze JavaScript code that had already been typed. In the end, however, we realized that converting every type into its own heap object was not a viable solution, because each object needs a allocation site, the amount of which has to be finite or the analysis loses the ability to terminate because a fixed-point no longer exists.

For the initial *global object*, this was not a problem as the initialization of the types does not contain a loop and therefore the allocation labels that need to be created in this step are finite by design.

For function calls that return objects, however, this is not the case. In the end we had to limit the use of the imported types to primitives, excluding objects and generics.

A possible solution might be to add another lattice member that is handled like object labels, but instead of storing its properties inside the heap its behavior on reads and updates is determined by a TypeScript type, namely the type it was created from on a return of a function. The introduction of those objects enables us to only use one of these per function call, limiting the amount of labels introduced into the analysis. This concept is currently not part of Inferium and has to be explored in future work.

3.11 Transfer functions

When we perform the data-flow analysis on the graph, each node has by design only one incoming edge — with the exception of merge nodes. When processed, The incoming state is then transformed into a new state, which is passed to further connected nodes. There are three special cases:

- merge nodes are the only node type that can have multiple incoming states. They join these together and pass them to the succeeding nodes.
- cond nodes have two successor nodes (t_1 and t_2). When the *condition* for the jump is truthy, only t_1 will be analyzed. If it is falsy, only t_2 will get control flow. If the condition could be both, t_1 as well as t_2 are analyzed (For more information on conditional jumps see 3.12).
- If the procession of a node results in an exception, the control flow is not given to the next regular node, but instead to the node that is responsible for handling exceptions.

As an example, we describe the transformation process of a writeP node for a property name $s \in S$, with an assignment site a and an execution state $e = (stack, heap, this, lexFrames)$.

1. Take value v and value o from the stack of the incoming state so that $v : o : rest = stack$. v will be the value we write into the property s on the objects pointed to by o.
2. Let $o' = normalize(o)$.
3. Let $ty = toTS_{NormValue}(o')$.
4. Coerce o' to a set of object labels $O \subseteq L$. The coercion for object labels in o is trivial. For the primitive values we have special predefined objects that model `Boolean`, `Number`, and `String` that we add to O if needed. All probes PS as well as $undef$ and $null$ are ignored in this step.

5. If o contains $null$ or $undef$, throw an exception by passing the state $e' = (exp, heap, this, lexFrames)$ with an exception object exp to the merge node that is responsible for handling exceptions at this point.

6. If O and PS are both empty, stop processing and do not pass any control flow to the next node.

7. If $|O| + |PS| = 1$, set $isStrongUpdate$ to $true$, and to $false$ otherwise.

8. Let $(h^*, h^@, ass) = heap$

9. For each $l \in L$

   a) If $l \in L^@$, let $(sites, configurable, enumerable, writable) = h^@(l)(s)$. If $writable \sqsupseteq write$ update $h^@(l)(s) = (X, configurable, enumerable, writable)$. Where X is $\{a\}$ if $isStrongUpdate$ is $true$ and $\{a\} \sqcup sites$ if $isStrongUpdate$ is $false$.

   b) If $l \in L^*$, let $(value, configurable, enumerable, writable, isAbsent) = h^*(l)(s)$. If $writable \sqsupseteq write$ update $h^*(l)(s) = (value \sqcup o', configurable, enumerable, writable, isAbsent)$.

10. For each probe $p \in PS$ add the constraint $write(p, s, ty)$ to the global constraint system.

11. Update $ass(a) = v$

12. Use the updated versions of $h^@$, h^* and ass to build new heap $heap' = (h^*, h^@, ass)$.

13. Pass new execution state $e'' = (o : rest, heap', this, lexFrames)$ to the next node.

3.12 Assignment site filtering

Earlier, we introduced assignment sites, which enable us to filter values that have been written to properties or local variables.

In the following section we will present the filtering algorithm. It is used when a cond node is processed.

The cond node has two target nodes. t_1 for the case that the condition c is truthy and t_2 for the case that the condition is falsy.

When we determine that the condition can be truthy as well as falsy, we have to analyze both, t_1 and t_2. Instead of handing the exact same state to both nodes, we filter all assignment sites that we can reach through the reference in condition c, so that the remaining values in the reached assignment site have the expected value.

Figure 3.2 shows the pseudo code for the filtering of a value. expected would be *truthy* for the then branch and *falsy* for the else branch.

```
1  let marked_sites = {}
2
3  count_expected(value, heap, expected)
4      count = 0
5      let (u,nl,b,n,s,os,ps,refs) = value
6      for (c ∈ {u, nl, b, n} ∪ s ∪ os ∪ ps ∪ refs)
7          if (c couldBe_heap expected)
8              count += 1
9      return count
10
11 mark_site(site, heap, expected, visited)
12     if (site ∈ visited)
13         return
14
15     let value = heap_Assignment(site)
16     if (mark_value(value, heap, expected, visited ∪ {site}))
17         marked_sites ⊔= site
18
19 mark_value(value, heap, expected, visited)
20     let n = count_expected(value, heap, expected)
21     if (n > 1)
22         return false
23
24     let (_,_,_,_,_,_,_,refs) = value
25     for (ref ∈ refs ∧ ref couldBe_heap expected)
26         mark_site(ref, heap, expected, visited)
27     return true
28
29 filter_condition(value, heap, expected)
30     mark_value(value, heap, expected, {})
31
32     for (s ∈ marked_sites)
33         remove all elements from heap_Assignment(s) that can not be expected
```

Figure 3.2: Algorithm for the filtering a condition.
filter_condition takes a value and filters all values that can not be expected. expected is either *truthy* or *falsy*. If an assignment site can be expected because of multiple $Value$ components, we can not conclude anything about the site. $couldBe_h$ checks for a member of the $Value$ lattice if it might be *truthy* or *falsy* depending on expected. References are resolved via heap h and recursively checked for their value.

3.13 Function calls

In JavaScript, calls have a complex semantic, which we can model precisely with our analysis. When a call node is processed, it pops a value o from the stack of the incoming state, normalizes it, and filters it for object labels $O \subseteq F$, that have an internal function descriptor attached. This descriptor decides how the function object is processed.

Built-in functions Built-in functions are those that are defined by the core analysis' code itself. The built-in function gets handed the arguments that the function was called with and the heap of the incoming state. It performs its internal functionality and returns a new heap and a return value, which are both used in the execution state for the next node.

Currently the only built-in function that Inferium supports is require, a function defined by the Node.js module system and which is used to load modules from within another module.

Signature functions For externally loaded types of functions, we attach a *signature function descriptor* to the function label. It holds the type of the function as it was defined in the type definition. The idea was to transform the arguments into TypeScript types and perform a full type inference with the function call. The inferred return type could have then been converted into a lattice member and used in the analysis. The full type inference algorithm of TypeScript was out of scope for this project, though, and so we currently only perform a minor type check and retire to any as return value if we encounter more complex types.

Code-defined functions If the JavaScript code defines functions itself, their internal code is also converted into the graph representation. Graphs for functions always have an *entry node* that is the internal function descriptor of the allocation site's label. If the call node encounters such a label, it gets this entry node and directs control flow the node. As said before, the *AnalysisLattice* describes the state of the analysis as a whole by assigning a state to each node. In fact, it assigns multiple states to each node, as a version name $v \in V$ models the call context of the state in the node. When we perform a function call with the function label f at call node n where the incoming state was assigned in the *AnalysisLattice* by $v \times n \mapsto s$, we do not pass control flow to the same v but we add (f, n) to create a new version name. With this our analysis becomes context-sensitive.

Now, recursion becomes a problem, because it might lead to an ever growing call context. We counter this by checking if the called function f is already part of the call context. If so, we have detected recursion at node n_1 with version name v where n_2 is the call node that called f before. Instead of creating a new version name, we pass control flow to the node that n_2 passed control

flow to when it called f. To have an execution state for the node after n_1, we simply take the outgoing state of n_2.

Additionally, we have to handle probes that might be in o. The implication of a probe in this context is that a user-given function was called. We record this information by adding a *call* constraints to the global constraint system. For every probe $p \in o$, with the to TypeScript types converted arguments tys, and the return probe r, we add $call(p, tys, r)$ to the global constraint system.

The return probe r exists once per call node and models the return value of the user function, about which we know nothing. It enables the analysis to gather information about the subsequent use of the return value. Because we bound it to the probes in o with the *call* constraint, we can retrieve it later from the constraint system and extract information about the return type of the probes in o.

3.14 Analyzing packages

After having described the data-flow analysis, in this section we will show how exactly it is used in Inferium.

When analyzing a JavaScript package, one major information is missing: how the package is used by the user. We therefore can not perform a whole program analysis but have to analyze the module code and contained functions separately.

The first step Node.js performs when a JavaScript module is imported, is to execute its top-level code. This code does initialization and especially populates the export object. This is also the first step we perform.

We construct an initial heap that contains all the required objects and functions. We do this partly manually and partly automatically. The global object, for example, is solely constructed from the predefined types we imported, which was described in section 3.10.

This newly constructed heap is then used in the first execution state for entry node of the graph of the top-level module code. We run the analysis once and extract the resulting heap, which contains the exports object, that holds all the exported objects.

After this step we should have achieved similar results to *dts-gen* [3]. No functions have yet been analyzed, so we have no information about parameters and return types.

The heap we extracted from this initial phase would go to the user code, about which we know nothing. We call this heap the *user heap*. We expect, however, that the user code will call a function the package exported, so we have to analyze these functions.

To determine which functions we have to analyze, we examine the exports object and extract all function objects we can reach from it by recursively traversing the properties.

We then assign a probe to each parameter of the found functions and one by one analyze the functions' code with our analysis. As arguments we used the previously

assigned probes and the user heap is used as heap in the execution state. After each finished analysis, we join the resulting heap with the user heap to build the new user heap for the next function.

Additionally, we examine the returned value for further functions. These functions could again be called by the user and have to be analyzed.

Note that the top-level module code is only analyzed once, which models that it is only executed by Node.js once the module is imported.

We repeat this process until we find a fixed-point for the user state. This means, that none of the functions we have to analyze changes the user heap anymore.

We can now extract the type definitions from this user heap.

3.14.1 Termination

An important property of a data-flow analysis is that it terminates. The usual requirement for this would be that the lattice is of finite height and the transfer functions monotone. This, however, is not the case for our lattice as it contains multiple power sets and mappings. Like TAJS [5], we argue informally that the analysis terminates nonetheless because of the following observations.

- There are only a limited number of allocation sites and assignment sites if the set of possible version names is finite. This limits the height of the power set lattices for both site types as well as the height of the mapping lattices.
- Our analysis can handle an unlimited amount of different strings per value, and strings are not limited like sites are. We argue that there is still a finite number of strings for a given program, because each string has to be defined in the code. With this argumentation, we have to make sure that there is no operation in the analysis, that can combine strings into new ones. At the moment, this could only happen in the transfer function of a `binary` + node, so this operation has a check only to concatenate its two arguments if both are specific strings. Otherwise, the result is the $\top$ element for strings.
- The $ScopeChain$ is limited in size by the maximal nestedness of functions in the program code as it is not possible to programmatically extend the scope chain.
- The set of possible version names V must also be of finite size, because there are only a limited number of nodes N and a limited number of functions F and we prevent that the same $f \in F$ can occur twice in the same version name (see recursion in 3.13).

Together with the monotony of our transfer function, these observations ensure that a fixed-point can be found and that our data-flow analysis terminates.

Because the user heap is always joined with the resulting heaps of analyzing a function, the user heap only gets greater. Because $Heap$ has a finite height — as we argued above — a fixed-point can be found as well and the whole analysis terminates.

3.15 Generating type definitions

After we found a fixed-point for the user heap, we enter the final phase of Inferium and generate real TypeScript type definitions.

Because we are primarily interested in the type of the exports object, we can simply use $toTS_{Obj}$ to transform our lattice representation into a TypeScript type. The printing of these types is trivial, but we also introduced the probe types, into which $toTS_{NormValue}$ converts probes.

These probe types can each be printed as its own generic type, but we also need an object or a function type where we can declare the generic type variable on. There we can also print out their upper bound, which is defined by constraints in the global constraint system. A type ϱ_p for probe p with the constraint $useAs(p, number)$ has the upper bound number, while a constraint $read(p, "x", p_2)$ would upper-bound ϱ_p to an object type with a property x which itself has the type ϱ_{p_2}.

Additionally, some of these generics are superfluous. In the following example, T is not needed and could be replaced by its upper bound.

```
function test<T extends number>(p:T): void
```

The upper bound $\beta(p)$ for the probe p, is extracted from the constraints with the following rules.

$$
\begin{aligned}
write(p_1, s, ty) &\quad \mapsto \beta(p_1) \sqsubseteq \{s?\colon ty\} \\
read(p_1, s, p_2) &\quad \mapsto \beta(p_1) \sqsubseteq \{s\colon \varrho_{p_2}\} \\
useAs(p_1, ty) &\quad \mapsto \beta(p_1) \sqsubseteq ty \\
call(p_1, tys, p_2) &\quad \mapsto \beta(p_1) \sqsubseteq tys \to \varrho_{p_2} \\
new(p_1, tys, p_2) &\quad \mapsto \beta(p_1) \sqsubseteq new(tys \to \varrho_{p_2})
\end{aligned}
$$

Note that the $write$ constraint forces $\beta(p_1)$ to accept a property s with type ty. This is because $\beta(p_1)$ only needs to accept that its property s gets written. It does not have to provide a value when s is read. The question mark indicates that, the property is allowed to be absent.

Because we create a new probe for every node that performs a property read, we have multiple probes per property name that have $read$ constraints with the same probe p. Take the following code, for example.

```
function f(obj) {
    return obj.prop - obj.prop
}
```

When `obj` has the probe p_1, the left `obj.prop` results in p_2 and the right `obj.prop` in p_3, then the following constraints are recorded.

$$read(p_1, "prop", p_2)$$

$$read(p_1, "prop", p_3)$$

$$useAs(p_2, number)$$

$$useAs(p_3, number)$$

Because of the definition of β, p_2 and p_3 get intersected in the TypeScript type system. To reduce the number of probe types, we use a union-find algorithm to merge all probe types $p_1, ..., p_n$ that occur in the same intersection type $p_1 \sqcap ... \sqcap p_n$ and replace them with a new probe type $p_\sqcap$. The upper bound of $p_\sqcap$ is the intersection of all upper bounds from the original probe types: $\beta(p_\sqcap) = \beta(p_1) \sqcap ... \sqcap \beta(p_n)$

Before we can print the type definitions, we have to determine which objects and which functions are going to have which generic parameters. The probe types essentially are the generic types but are currently unbound. In the last step we have to find those objects and functions that bind these generic types.

We do this by recursively traversing all object types, beginning with the exports object, and looking at which probe types are used in inner types. For object types, if a probe type can be reached through two or more properties, we also bind the respective generic type as generic parameter. For functions, we look into the parameters and the return type. Additionally, we also bind those generic variables that are used in direct types of parameters.

The following example demonstrates this.

```
function create() {
    var store
    return {
        set: function(x) { store = x },
        get: function() { return x }
    }
}
```

After the analysis, we know that the probe type p, that was used as argument for `set`, flows to `store` and then to the return of `get`. The resulting type definition with the unbound p looks as follows.

```
interface I {
  set(x: p): undefined
  get(): p
}

function create() : I
```

Now we look into `I` and see that p is used in `get` as well as in `set`. So we bind p to `I`. Because `I` is used in `create` and `create` is a function, we also bind the variable there. For `T` as generic type name for p, the result looks like follows.

```
interface I<T> {
  set(x: T): undefined
  get(): T
}

function create<T>() : I<T>
```

This procedure, however, does not work in some situations where probes flow into internal variables that are further outside of the scope than they are exposed to the API by. The following, slightly changed example shows the problem.

```
var store
function create() {
    return {
        set: function(x) { store = x },
        get: function() { return x }
    }
}
```

This example yields the exact same type definition, which is not correct. Because `store` is a global variable, it can not be generically typed but must be set to its upper bound, which is `any`.

A possible solution to this problem would be to save all probe types that are in local variables of a function in the function's type. When a function has a local variable with a probe type the respective generic type must also be bound to the function's type.

4 Implementation

Inferium employs a multi-stage process to generate type definitions for a given Node.js package. Given the name of a target package we perform the following steps:

1. We use the Node.js' package manager npm to download the target package and its dependencies into a temporary directory. Existing type definitions for these packages are downloaded too.

2. We generate a simple TypeScript file that imports all dependencies of the target package. This file is then given to the TypeScript compiler who infers types for the exports of all dependencies, that already have type definitions. Afterwards we can use the API of the compiler to extract these type definitions and print them into a single json file. The standard library of TypeScript alone, which models the base JavaScript functions and the global object, yields around 1,200 interface and function types.

3. The resulting json file is then read by the core program of Inferium and used in the analysis of the target package. When a fixed-point is found for the user state, the parameters and return values of all user-accessible functions are analyzed for their types and printed into a type definition file. Both, analysis and printing, are implemented in around 8,000 lines of Scala code.

In the following section, we will discuss the implementation of the Heap, which has to store all existing objects and assignment site mappings.

4.1 Heap

One important component of the analysis is the execution state, which is modeled as described in section 3.8. $Stack$, $This$, and $ScopeChain$ have a low memory footprint and are efficiently implemented using immutable shared data structures, which are part of the Scala Standard Library [9].

In principle, the heap is nothing more than a map from allocation sites to object descriptors and a map from assignment sites to values, which would predestine it for an immutable map, which is already available in Scala. Unfortunately, we have to join states at merge nodes, regularly. An operation that is costly, because the map has no further information about which mappings have changed. With around 1,000 objects at the beginning of the analysis, this would result in a merge of thousands of objects after every simple if statement.

To improve the merging capabilities, we use a heap implementation that is based on the observation that only those objects have to be merged which were updated since the last common node.

```
var a = { p: "x" }
var b = { p: "y" }

if (Math.random()) {
  b.p = true
} else {
  b.p = false
}
```

In the above code for example, we only have to merge the object `b` points to at the merge node after the if statement. The object `a` points to has not been updated since the branching.

Instead of using a single mapping of allocation-sites to objects as well as a single mapping for the assignment sites, we use a chain of *heap chain elements* where each element e has mappings for objects and assignment sites as well as a parent heap chain element $p_1(e)$.

When we want to access a certain object $obj_h(l)$ with the allocation site l in heap h, we look if the object mapping m_e of the current chain element $e = elem(h)$ is defined for l. If a mapping exists, we use the mapped object. If not, we look into the heap element's parent $p_1(e)$ and so on until we find a mapping for l in the nth parent $p_n(e)$.

$$obj_h(l) = obj(l, elem(h))$$

$$obj(l, e) = \begin{cases} m_e(l) & \text{if } l \text{ is defined on } m_e, \\ m_{p(e)}(l) & \text{otherwise} \end{cases}$$

If we want to update an object referred by l on a heap h, we use the same procedure to get the abstract object $o = obj_h(l)$, perform the update on o to get o', update the mapping m of $e = elem(h)$ to get m', where $m'(l) = o'$, and create a new heap element e' which has the same parent as e and use it in a new heap h'. This way we get o' when we try to access $obj_{h'}(l)$.

Note that each heap chain element has not only two mappings for the abstract objects, but another one for the assignment sites.

When we want to join two heap elements e_1 and e_2, we can simply find the most recent common ancestor $c(e_1, e_2)$ of e_1 and e_2. To perform the actual join we have to transform both heap elements into new elements e_1' and e_2' so that $p_1(e_1') = p_1(e_2')$. This transformation is done by merging the mappings of a heap element with that of its parents using *squash* until the next parent is the common ancestor.

$$squash\colon (a \hookrightarrow b) \mapsto (a \hookrightarrow b) \to (a \hookrightarrow b)$$

$$squash(m, m_p) = l \mapsto \begin{cases} m(l) & \text{if } l \text{ is defined on } m, \\ m_p(l) & \text{otherwise} \end{cases}$$

After we have calculated e'_1 and e'_2, we can join their mappings m'_1 and m'_2 to create a new heap element $e_\sqcup$ with mapping $m_\sqcup$.

$$m_\sqcup = l \mapsto (obj(l, e_1) \sqcup obj(l, e_1))$$

5 Evaluation

To our knowledge, Inferium is the only tool that can generates TypeScript type definitions for real world Node.js packages and that generates more specific types than dts-gen [3], which sets all parameter and return types to any.

Our goal was to generate TypeScript type definitions that are helpful to a programmer who uses a Node.js package that was not typed. The definition of helpful is key to evaluate the performance of Inferium.

Because TypeScript type definitions are not meant to be fully correct, it becomes even harder to define helpfulness. There are always constructs that can not fully be typed even if TypeScript is continually updated to enable more fine tuned definitions.

The only point of reference are existing type definitions that were written by hand with the intent to be helpful.

To evaluate Inferium, we analyzed npm packages that had already been typed by hand and of which the type definitions were publicly available. We than compared the results of Inferium with those hand written type definitions.

As Inferium currently prints type definitions in a strict clinical way, we are not interested in the syntactical structure of the generated type definitions but rather in the generated types themselves.

We use the following indicators to evaluate our generated type definitions.

Exported Properties The percentage of properties that are directly on the exports object or transitively reachable from it. Here other tools ([4], [3]) reach usually a high percentage, because they execute the Node.js package and examine the resulting exports object instead of analyzing the top-level module code like Inferium does.

Expected Properties The percentage of properties that are part of types on function parameters. These are not easily obtainable by scanning the exports object, because their existence is only given implicitly by the usage of the parameters. Inferium uses probes to find them (3.7).

Provided Properties The percentage of properties that are part of function return types. Like Expected Properties, these are only defined implicitly in the JavaScript code and have to be found by analyzing functions.

Correct Types The percentage of property types that were correctly inferred by Inferium.

Bloated Generics The percentage of functions that have overly extensive generic type parameters. Inferium has the characteristic to generate too many type parameters for certain functions. With this we want to count the severity of this problem.

Internals The number of additional internal properties that have been found.

We used 12 small packages to apply our indicators. The results are shown in figure 5.3. Inferring types for the exports object works quite well, as the top-level module code often only consist of populating the exports object.

For the types themselves, a lot of precision is lost in the type generation phase. While the data-flow analysis was keeping the objects separate from each other, the type generation has the task to merge them together to produce readable types. As the merging algorithm that is currently used in Inferium uses a lot of approximation, much information is lost.

On the other hand, a lot of internal properties are exposed by Inferium. Because JavaScript does not know the concepts of private properties and we have no heuristic to classify them, they appear in the type definition — here most notably in *animation-frame*.

Most object types that where inferred for parameters (ExpP) where actually supertypes of string or array. When `forEach` is called on a parameter, a type for `forEach` is inferred, but there is no other hint, that the parameter's type should actually be array. We would need further heuristics to make this connection.

Three notable packages are *bezier–easing*, *methods*, and *ski*.

The definition for *bezier–easing* is shown in 5.1. The idea of the package is to provide two coordinates and return a function, that eases between $(0, 0)$ and $(1, 1)$ using a bezier-curve according to the provided coordinates. In the type definitions, this is the interface I40, which correctly takes a number and returns a number. In the case that the two provided coordinates are both on the line between $(0, 0)$ and $(1, 1)$, however, a simple identity function, that models the linear function, is returned. Inferium will infer different types for both functions and rightfully turn out with a generic identity function as interface I41 is. Rightful as it may be, it is not helpful, because it exposes implementation details to the package user.

methods has the same problem. It exports an array of strings, but Inferium exposes the details of this array including the exact mapping of the 28 values.

For some packages like *ski* the dataflow approach works remarkably well and gathers the correct constraints to correctly type the functions of the SKI calculus [10] as shown in figure 5.2.

```
interface I40 {
   (x: number): number
}
interface I41 {
   <T6>(x: T6): T6
}
interface I39 {
   (mX1: number, mY1: number, mX2: number, mY2: number): I40 |
       I41
}
declare var expr: I39
export = expr
```

Figure 5.1: Type definitions for the package *bezier–easing*.

```
export declare function S<T, S, U>
       (x: (z: U) => (y: S) => T, y: (z: U) => S, z: U): T;
export declare function K<T, S>(x: T): (y?: S) => T
export declare function I<T>(x: T): T;
```

a Original type definitions form *DefinitivelyTyped*

```
interface I13<T1, T4> {
  (_?: T4): T1
}
interface I12<T1, T4, T8> {
  (_?: T8): I13<T1, T4>
}
interface I14<T4, T8> {
  (_?: T8): T4
}
interface I15<T9> {
  (): T9
}
interface I11 {
   S<T1, T4, T8>
            (x: I12<T1, T4, T8>, y: I14<T4, T8>, z: T8): T1
   K<T9>(x: T9): I15<T9>
   I<T10>(x: T10): T10
}
declare var expr: I11
export = expr
```

b Correctly generated type definitions

Figure 5.2: Type definitions for the package *ski*.

Package	ExpP	ExpC	PP	CT	BG	I
absolute	0% (1)	-	-	0% (2)	0	0
animation-frame	0% (1)	0% (1)	-	0% (1)	0	12
base64-js	100% (3)	-	-	100% (2)	0	0
bech32	100% (4)	-	100% (2)	83% (12)	2	0
bezier–easing	100% (1)	-	100% (1)	83% (6)	0	0
btoa	100%	-	-	100% (2)	1	0
exit	100% (1)	-	-	100% (2)	0	0
fresh	100% (1)	-	-	66% (3)	0	0
methods	100% (1)	-	-	0% (1)	0	0
pure-render-decorator	100% (1)	-	-	100% (1)	1	0
sanitize-filename	100 % (1)	100 % (1)	0% (1)	75% (4)	1	0
ski	100% (3)	100% (1)	-	100% (8)	0	0

Figure 5.3: The results of the evaluation
From left to right: The package name (Package), Exported Properties (ExpP), Expected Properties (ExcP), Provided Properties (PP), Correct Types (CT), Bloated Generics (BG), Internals (I).

6 Related Work

As JavaScript is a popular programming language some other projects have already explored the possibility of statically analyzing JavaScript code.

With TAJS a framework for statically analyzing JavaScript code exists that is even able to analyze large JavaScript libraries [5]. The project transforms JavaScript into an assembler-like representation and performs a flow-sensitive, context-sensitive data flow analysis. From the values that are found in variables and properties, simple types can be inferred. However, this does only work for functions that are called when analyzing the top-level module code.

No information is won about parameters and return types that are not reachable in the call graph. Andreasen et al. [11] shows that even many complex JavaScript idioms can be analyzed by static analysis.

Feldthaus et al. [12] introduce TSCHECK, a tool that takes existing type definitions and checks if they apply to the targeted JavaScript implementation. For this, the code is executed with a JavaScript engine and the resulting state of the engine is compared to the type definition.

Next to the structural equality, which checks the existence of classes and functions, TSCHECK also starts a deeper analysis of the code inside of the functions. Even generic types are checked.

TSINFER [4] is another tool that tries to generate type definitions from JavaScript programs.

It also uses a data flow analysis to compute the values of variables and properties, but in a two step manner.

First, it starts a top-down analysis, to see which values flow into which variables. Afterwards, it reverses the process and uses a bottom-up data-flow analysis to see what type requirements different variables have. Their analysis is flow-insensitive and only produces simple types. No generics are inferred.

Chandra et al. [13] presents a tool, that analyzes JavaScript programs and infers types to compile the code into a static program. They are also able to compile libraries that include uncalled functions.

To analyze these functions they construct a set of constraints from the abstract syntax tree and solve it. Generic types are only available for handwritten types and cannot be inferred.

Originally, there were no explicit classes in JavaScript, and classes had to be simulated by functions and the prototype chain.

Today JavaScript has a syntactical construct that mimics classes, even if the underlying semantic still uses prototypes and functions.

To provide class information about code that doesn't contain explicitly defined classes, Eshkevari et al. [14] investigate ways to recognize classes and inheritance relations from such code.

There are also other languages that support extensive type systems. Chaudhuri et al. [15] present the type system of *Flow*. Similarly to TypeScript, Flow offers the option to annotate variables with types. For the check, a module-wide constraint graph is constructed and checked for violations of certain rules. Especially, type narrowing is built in well. Types can change in contexts where they can only be certain values, as the following example demonstrates.

```
function test(x: number | null) {
  if (x)
    // x cannot be null and was narrowed to number
    return x - 3
  return 0
}
```

At first glance, it also looks as if Flow could infer even generic types, but it cannot. Also functions are handled context-insensitive in the constraint graph.

7 Conclusion

Statically analyzing JavaScript is problem to begin with, but is even harder for functions that are never called. Not knowing what values can occur, limits the ability of the analysis.

With *probes* we showed that it is possible to extend the lattice of a data-flow analysis to accommodate elements that model incoming values of arguments. We were able to comprehend the flow of user input through the program and generate even generic types for functions and interfaces.

With the introduction of *assignment-site abstraction*, we showed a way to model path-sensitivity in a data-flow analysis that can handle updates of variables and properties, even if the missing strong update capabilities of Inferium prevented the concept from having any real impact on the results.

Working with TypeScript's type system proofed to be very challenging. Because TypeScript's goal is to provide types for as many common JavaScript patterns as possible, its type system is more extensive than that of other programming languages. Union types, intersection types, and generics are a complex mix to handle and we didn't even consider the conditional types that were introduced by TypeScript 2.8 [16]. Bringing all these types into the analysis was a hard problem, and the approach we chose of converting types into lattice members, produced more problems than it solved.

We still believe that a data-flow analysis is the correct path to gather precise type information for dynamic languages like JavaScript.

There is a long way to go until Inferium or any other tool will be able to generate really helpful type definitions. Nevertheless, we believe that Inferium provides a good basis to build upon for a JavaScript type inference.

7.1 Future Work

As the results show, Inferium has a lot of potential for improvement. The basic idea is working, but the details have to be refined, before the results are anywhere near helpful to programmers.

There are three major parts in Inferium that need improvement: the handling of predefined types, the data-flow analysis itself, and the type generation.

For the predefined types, we chose an approach where we converted the types into lattice members to bring them into the analysis. This worked fine for the initial existing global object but failed when we tried to convert return types into lattice

members because objects are bound to allocation sites, of which only a finite number are allowed to exist so that the analysis terminates. Future work has to find a way to handle these types inside of the analysis. Maybe, a possibility would be to introduce a new lattice member to $NormValue$ that represents objects that where only defined as types. Because a join on of them would result in the join of the underlying TypeScript types, no allocation sites would be needed.

For the data-flow analysis, the most important improvement that is needed, are *strong updates* for more cases than what recency abstraction solves. In the following code

```
var obj = Math.random() > 0.5? {} : {}
obj.prop = "Hello"
console.log(obj.prop)
```

Inferium fails to perform a strong update on `obj.prop` because it doesn't know exactly to which object `obj` refers. Even if it is ambiguous to which object `obj` points, we know that after `obj` has been assigned, in subsequent usages it always refers to the *same* object.

We see an opportunity in constructing the *single static assignment* form of the program and insert proxy objects at assignments. These proxy objects would summarize underlying objects but could handle strong updates.

With the introduction of the *assignment-site abstraction* we showed how to bring path-sensitivity into our data-flow analysis. This idea is underdeveloped in Inferium and there is a lot of room for improvement. The filtering of variables is a nice start, but we imagine much more complex references, to handle filtering with expressions that involve `typeof` or even function calls. It has to be determined if such references are possible in a data-flow analysis.

A very important feature of TypeScript are overloaded functions. They provide a facility to model dependencies between parameters and return types. To infer them is a harder problem, because we would need precise information under which conditions values can occur in variables and properties. We believe that *assignment sites* are the right place to attach these conditions. How they are created, handled, and later converted into types has to be determined.

Lastly, the handling of the TypeScript's type system in Inferium has to be improved to generate more exact and correct types. Especially, the conditions under which types should be merged needs more investigation.

We have not found a way to use the TypeScript compiler for this purpose, because of the additional *probe types* that we have to handle, but if there is a way, this would free Inferium from keeping up with the ever growing type system of TypeScript.

Bibliography

[1] "The redmonk programming language rankings: January 2018." https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/. Accessed: 2018-08-01.

[2] "Module counts." http://www.modulecounts.com/. Accessed: 2018-08-01.

[3] "dts-gen repositiory." https://github.com/Microsoft/dts-gen/. Accessed: 2018-08-02.

[4] E. K. Kristensen and A. Møller, "Inference and evolution of typescript declaration files," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 99–115, Springer, 2017.

[5] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, pp. 238–255, Springer, 2009.

[6] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," *SIGPLAN Not.*, vol. 25, pp. 296–310, June 1990.

[7] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, (New York, NY, USA), pp. 66–74, ACM, 1982.

[8] G. Balakrishnan and T. Reps, "Recency-abstraction for heap-allocated storage," in *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, (Berlin, Heidelberg), pp. 221–239, Springer-Verlag, 2006.

[9] "Concrete immutable collection classes." https://docs.scala-lang.org/overviews/collections/concrete-immutable-collection-classes.html. Accessed: 2018-07-28.

[10] M. Schönfinkel, "Über die bausteine der mathematischen logik," *Mathematische Annalen*, vol. 92, pp. 305–316, Sep 1924.

[11] E. Andreasen and A. Møller, "Determinacy in static analysis for jquery," in *ACM SIGPLAN Notices*, vol. 49, pp. 17–31, ACM, 2014.

[12] A. Feldthaus and A. Møller, "Checking correctness of typescript interfaces for javascript libraries," in *ACM SIGPLAN Notices*, vol. 49, pp. 1–16, ACM, 2014.

[13] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 410–429, ACM, 2016.

[14] L. Eshkevari, D. Mazinanian, S. Rostami, and N. Tsantalis, "Jsdeodorant: Class-awareness for javascript programs," in *Proceedings of the 39th International Conference on Software Engineering Companion*, pp. 71–74, IEEE Press, 2017.

[15] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, "Fast and precise type checking for javascript," *arXiv preprint arXiv:1708.08021*, 2017.

[16] "Typescript 2.8 announcement." `https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-8.html`. Accessed: 2018-08-01.

Erklärung

Hiermit erkläre ich, Tobias Kahlert, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

__________________	______________________________
Ort, Datum	Unterschrift