IMPERIAL COLLEGE LONDON

MASTER'S THESIS

Typing JavaScript through Symbolic Execution

Author:
Radu-Andrei SZASZ

Supervisor:
Prof. Philippa GARDNER

A thesis submitted in fulfillment of the requirements for the degree of Masters in Engineering

in the

Department of Computing

June 18, 2018

IMPERIAL COLLEGE LONDON

Abstract

Faculty of Engineering

Department of Computing

Masters in Engineering

Typing JavaScript through Symbolic Execution

by Radu-Andrei SZASZ

JavaScript is the de facto language for client-side web applications, supported by all major browsers. It is also used to build server-side solutions and mobile applications. Despite its massive popularity, JavaScript's dynamic nature, together with its complicated semantics, make it a troublesome language for developing and maintaining large applications. In the past decade, there have been numerous attempts to retrofit a static type system on top of JavaScript. The most popular such extension is TypeScript.

This work builds a mechanism for using TypeScript type annotations as light-weight specifications for a static analysis tool based on Separation Logic, JaVerT. We provide a sound translation from TypeScript type annotations to JaVerT assertions and implement a tool that carries out this translation. The translation provides strong guarantees for `class` types, ensuring prototype safety and enabling reasoning about scope chains. We applied our approach to a number of examples, where we were able to achieve a 40-fold reduction in the specification character count.

Contents

Chapter 1

Introduction

JavaScript is the de facto language for client-side web applications, supported by all major browsers. It is also used to build server-side solutions and mobile applications. Despite its massive popularity, JavaScript's dynamic nature, together with its complicated semantics, make it a troublesome language for developing and maintaining large applications. This drawback is amplified by the lack of adequate static analysis tools.

In the past decade, there have been numerous attempts to retrofit a static type system on top of JavaScript. The most popular such extension is TypeScript [6], with over 200.000 GitHub repositories using it as of 2017 [23]. It is an extension of JavaScript that enriches it with interfaces and a static type system. TypeScript improved the JavaScript developer experience greatly, by enabling a streamlined IDE experience previously associated with languages such as Java or Scala. The main disadvantage of TypeScript is that its type system is unsound by design and hence type checking does not provide the safety developers expect. Flow [9], an alternative to TypeScript, tracks the flow of values to produce a sound type system for a large subset of JavaScript. If we consider the entire JavaScript language, Flow can not guarantee type safety either.

In contrast, Fragoso Santos et al. developed JaVerT [17], a JavaScript verification toolchain based on separation logic. The authors of JaVerT provide an assertion language, JS Logic, which is required for writing the specifications of JavaScript code. Programmers need to specify function pre- and post-conditions, loop invariants and instructions for folding and unfolding user-defined predicates. The code annotated with JS Logic assertions is then compiled to JSIL code annotated with JSIL Logic assertions. JSIL is an intermediate goto language capturing the dynamism of JavaScript. Lastly, the toolchain contains a semi-automatic verification tool named JSIL Verify, which performs verification at the JSIL level. These elements enable programmers to verify functional correctness properties of their code. The main drawback of this approach is that developers are required to annotate their methods with JS Logic assertions and instructions for carrying out the proof, which is often as complicated as writing the code in the first place.

The goal of this project is to combine the approaches described above. More specifically, we process JavaScript programs containing TypeScript type annotations and translate the type information into JS Logic assertions. The mechanical checking of the JavaScript code annotated with JS Logic assertions is then facilitated by JaVerT. This approach provides users with the intuitive interface of TypeScript type annotations, while offering them the precise analysis of JaVerT. Soundness is a core goal of the resulting system and a proof of it is presented in this report. Common JavaScript idioms should be supported and the code should be interpreted as dictated by the JavaScript standard.

1.1 Related work

We present a timeline of the work done with the goal of retrofitting a type system on top of JavaScript. This culminates with the development of TypeScript and Flow.

1995	JavaScript is launched and deployed in Netscape Navigator 2.0 beta 3.
2005	Thiemann presents a first attempt at defining a type system for JavaScript [40]. The system tracks the possible traits of an object and flags suspicious type conversions. The work did not cover vital aspects of JavaScript, such as prototypal inheritance, and was not accompanied by an implementation at the time it was published.
2005	Anderson et al. [1] design a type system for JS_0, a subset of JavaScript, together with a sound type inference algorithm. This subset tracks dynamic addition/reassignment of fields and methods, but does not cover important features such as object indexing using strings and implicit coercions.
2009	Jensen et al.'s [25] work "is the first step towards a full-blown JavaScript program analyser". The analysis is built on previously developed techniques such as recency abstraction [2]. The analysis suffers from the presence of false positives and is infeasible for use in an IDE due to performance issues, as pointed out in the evaluation section of the paper.
2011	Alternative approach: Google launches Dart [7] — a typed programming language for client-side web applications that compiles to JavaScript.
2012	Microsoft launches TypeScript [31]. TypeScript is syntactically a superset of JavaScript. Its intuitive type annotations, performance and IDE support, coupled with the flexibility of its gradual type system were the factors behind the adoption process. A substantial drawback is the fact that the type analysis is unsound; sources of unsoundness include unchecked downcasts, unchecked gradual typing and unchecked covariance, all of which are covered at length in [6].
2014	In their work, Feldthaus et al. [15] check the correctness of TypeScript Interfaces for JavaScript libraries. Their system analyses the snapshot of the heap after library initialisation and performs a static analysis on the functions exported by the library checked. The mechanism is implemented in the toolchain `JSNAP`, `TSCORE` and `TSCHECK`.
2014	Facebook launches Flow [22]; its implementation is later described in [9]. Flow constitutes a system for both type inference and type checking. It analyses the flow of values, creating a constraint-graph and inferring types for the expressions in the program. Consequently, it requires far fewer annotations than TypeScript and presents fewer sources of unsoundness.
2015	Rastogi et al. [38] publish a paper describing a safe type system for TypeScript. They introduce a number of stricter static checks as well as runtime checks. The implementation minimises the performance hit taken by runtime checks by only instrumenting code that is not used in compliance with the static typing discipline.

Alongside with the work carried out to retrofit a type system onto JavaScript, it is important to note the contributions which led to the development of static analysis tools such as JaVerT.

1969	Hoare develops a formal system for reasoning about the correctness of computer programs [21]. He introduces a new notation: $P\{C\}Q$ to be interpreted as "If the assertion P is true before the initialisation of a program C, then the assertion Q will be true on its completion."
2001	O'Hearn et al. [35] create an extension of Hoare logic for reasoning about programs that alter data structures. The central piece of their work is the addition of the $*$ operator called *separating conjunction*, which enables reasoning about partial heaps. Intuitively, $P * Q$ describes the set of partial heaps that can be separated in two disjoint partial heaps, one satisfying P and the other one satisfying Q.
2005	Berdine et al. [5] develop a symbolic execution algorithm for automatically checking triples of the form $\{P\}C\{Q\}$ where P and Q are separation logic assertions. Their work is targeted at a C-like language for which they provide an operational semantics. In [4] the authors present an experimental tool built upon the algorithms in [5].
2008	Distefano and Parkinson create an automatic verification system for Java, jStar [12], abstract predicate families [36, 37] and the idea of symbolic execution and abstraction using separation logic. They use their system to verify four popular Java design patterns.
2011	Jacobs et al. [24] introduce VeriFast, a semi-automatic verification tool for C and Java. Verifast can verify both single- and multi-threaded programs, takes permissioning into account in the case of Java, allows user to define custom predicates, and provides predictable and fast verification times.
2012	Gardner et al. [19] adapt ideas from separation logic to provide a scalable program logic for a subset of JavaScript, based on an operational semantics faithful to ECMAScript 3. They model several challenging features of JavaScript, such as prototypal inheritance and scope chains.
2017	Fragoso Santos et al. [17] create JaVerT: a JavaScript verification toolchain, which allows for semi-automatic reasoning about functional correctness properties of JavaScript programs, without introducing simplifications to the language semantics. The annotation load required, however, is substantial.

1.2 Motivation

We illustrate two scenarios in which TypeScript fails to ensure type safety on its own, but we are able to enforce it by compiling TypeScript's type annotations to JS Logic assertions. Onward, we use the terms "JS Logic assertions" and "JaVerT assertions" interchangeably.

We first present the examples in JavaScript and illustrate some features of the language that we deem important. Next, in Figures 1.3 and 1.4, we present the corresponding

```
1  function createShape(shape, sz) {
2     if (shape === "circle") {
3        return { radius: sz };
4     }
5     else if (shape === "square") {
6        return { edgeSize: sz };
7     }
8  };
9  var shape = createShape("circle", 3);
10 var square = shape;
11 console.log(square.edgeSize); // undefined
```

FIGURE 1.1: A JavaScript example illustrating TypeScript's lack of support for flow sensitivity.

TypeScript code and introduce the syntax and semantics of TypeScript type annotations. Finally, in Figures 1.6 and 1.8, we show the JavaScript annotated with JaVerT assertions we compile the TypeScript code to and use these examples to introduce JaVerT assertions and explain why they can offer soundness where TypeScript fails.

The first of the two examples illustrates how type safety can be enforced by a flow-sensitive analysis. The second example highlights the benefits of using JS Logic assertions when dealing with dynamic property accesses—an area in which both TypeScript and Flow struggle.

1.2.1 Examples in JavaScript

Example 1: Flow sensitivity

In the example presented in Figure 1.1, we define a function that takes an argument `shape` and an argument `sz` and returns an object that describes either a circle or a square (lines 1-8). After the function call in line 9 we are free to assign, in line 10, the return value of the function, even though it describes a circle, to a variable called `square` that a developer would assume describes a square.

Not all paths in a JavaScript function need to return a value. If the end of a function is reached without encountering a `return` statement, the value `undefined` is returned by default. Another interesting point to make is that while `square = { radius: 3 }`, the access `square.edgeSize` returns `undefined` instead of triggering a runtime. This forgiving runtime behaviour enables the `undefined` value produced by accessing a missing field to propagate and cause a bug further in the execution flow; a bug occurring at a totally different place will be harder to trace and fix for a developer.

In an ideal world, a type system analysing the `createShape` function in a flow sensitive manner should detect that when called with the argument `shape` having value `"circle"`, the function returns an object with a single field named `radius`. Such a type system should complain that, in line 11, we are accessing a nonexistent field, thus saving a good amount of debugging time.

Example 2: Dynamic Property Access

In the code snippet presented in Figure 1.2 we implement a key/value map, `MyMap` (lines 1-14), which works by storing the key/value pairs into the `contents` object. We provide a constructor (lines 2-4) and two methods: the `get` method (lines 5-9), which returns the value corresponding to a key; and `put` (lines 10-12), which adds a key/value pair to the map, overwriting the old value if the key already existed.

```
1  var MyMap = (function () {
2    function MyMap() {
3      this.contents = {};
4    }
5    MyMap.prototype.get = function (k) {
6      if (this.contents.hasOwnProperty(k)) {
7        return this.contents[k];
8      }
9    };
10   MyMap.prototype.put = function (k, v) {
11     this.contents[k] = v;
12   };
13   return MyMap;
14 }());
15
16 var myMap = new MyMap();
17 myMap.put("myKey", 3);
18 myMap.put("hasOwnProperty", 0);
19 console.log(myMap.get("myKey"));
```

FIGURE 1.2: JavaScript example illustrating lack of type safety

Before covering the lines of code after the definition of MyMap, we need to explain two vital concepts: **prototype-based inheritance** and **prototype safety**.

Prototype-based inheritance. JavaScript does not model classes, but instead uses objects and prototypes. Each object has a private property holding a link to another object called prototype. The chain we get by following this link on an object is called *prototype chain*. The value of a property on an object is resolved by firstly inspecting the object itself and then traversing its prototype chain. This is the mechanism via which JavaScript models inheritance and is commonly referred to as *prototype-based inheritance*.

The prototype chain of nearly all JavaScript objects ends with a special prototype object, `Object.prototype` [33]. The `hasOwnProperty` method in `Object.prototype` checks whether an object has a certain property as its own property, as opposed to inheriting it. This method is called in line 6 to check that the key we are accessing was purposefully added to the map.

Prototype safety. In general, the specification of a given library must ensure that all prototype chains are consistent with correct library behaviour by stating which resources must not be present for its code to run correctly. In particular, (P1) constructed objects cannot redefine properties that are to be found in their prototypes; and (P2) prototypes cannot define as non-writable those properties that are to be present in their instances. We refer to these two criteria as *prototype safety*[1].

We illustrate how the example in Figure 1.2 breaks prototype safety. We first create a new key-value map (line 16) and associate the key `"myKey"` with the value `3`. Next, the call to `put` in line 18 shadows the `hasOwnProperty` method from `Object.prototype`, breaching condition (P2) of prototype safety. As (P2) is broken, the call to `get` in line 19 leads to a crash: when execution reaches line 6, `this.contents.hasOwnProperty` evaluates to `0`, and the call `0("myKey")` triggers a type error at runtime, as 0 is not a function.

We would like a type system that prohibits the assignment to the `hasOwnProperty` field, because it shadows the target method of `Object.prototype` and breaks prototype safety.

[1]Prototype safety as defined in [17]

```
1  interface Square {
2      edgeSize: number;
3  }
4
5  interface Circle {
6      radius: number;
7  }
8
9  function createShape(shape: "circle" | "square", sz: number): Square | Circle {
10     if (shape === "circle") {
11         return { radius: sz };
12     } else if (shape === "square") {
13         return { edgeSize: sz };
14     }
15 }
16
17 var shape: Square | Circle = createShape("circle", 3);
18 var square: Square = shape as Square;
19 console.log(square.edgeSize);
```

FIGURE 1.3: TypeScript code that transpiles to the JavaScript presented in Figure 1.1

The pattern used for object creation in the code in Figure 1.2 is common in JavaScript [10], as it enables information hiding via the use of closures.

1.2.2 Examples in TypeScript

TypeScript is an extension of JavaScript, adding interfaces and a static gradual type system to the language. Users interact with the type system via the use of type annotations which the TypeScript compiler completely erases once it finishes type checking the code—full type erasure is one of the characteristics of TypeScript. This implies that there exists no run-time representation of types and no run-time type checking; coupled with JavaScript's implicit type coercions, this characteristic of TypeScript is likely to allow bugs to propagate.

We present, in Figures 1.3 and 1.4, the TypeScript code that transpiles to the JavaScript code in Figures 1.1 and 1.2, respectively. We make use of these two examples to introduce the TypeScript type annotations; we give the full syntax of types in §2.1.

Example 1: Flow Sensitivity

The program presented in Figure 1.3 transpiles to the code we showed in Figure 1.1. We focus on the TypeScript-specific features present in this example. As a first observation, the TypeScript is much easier to grasp for someone coming from Java or C++ than the transpiled code.

TypeScript makes use of a structural type system; interfaces, declared by using the keyword `interface`, are used to describe the structure of an object. The declaration in lines 1-3 describes a `Square`. A `Square` must have a field `edgeSize` of type `number`. Similarly, a `Circle` (lines 5-7) must have a field `radius` of type `number`.

Focusing on the type annotations present in the definition of `createShape`, we notice that the function takes two arguments: `shape`, of type `"circle" | "square"`, which is called a string literal union type [26]; and `sz` of type `number`. Intuitively, the type annotation of `shape` says that `shape` must represent either the string `"circle"`, or the

```
1  class MyMap {
2    private contents: { [key: string]: number };
3
4    public constructor() {
5      this.contents = {}
6    }
7
8    public get(k: string): number | undefined {
9      if (this.contents.hasOwnProperty(k)) {
10         return this.contents[k];
11     }
12   }
13
14   public put(k: string, v: number): void {
15     this.contents[k] = v;
16   }
17 }
18
19 var myMap = new MyMap();
20 myMap.put("myKey", 3);
21 myMap.put("hasOwnProperty", 0);
22 console.log(myMap.get("myKey"));
```

FIGURE 1.4: TypeScript code that transpiles to the JavaScript presented Figure 1.2

string `"square"`. The `|` symbol constructs a union type. Another example of such a union type is present in the return type of the function, `Square | Circle`, meaning that the function will return either a `Square` or a `Circle`.

The call in line 17 creates a `Circle`. TypeScript's type analysis is not flow sensitive, so it cannot determine this; the variable `shape` must have type `Square | Circle`. In line 18, we are downcasting `shape` to a `Square` and assigning it to the variable `square`. TypeScript is unable to statically verify whether or not this downcast is valid and, hence, does not throw a compile time warning. Because of full type erasure, checking the validity of the downcast cannot be performed at runtime either. A consequence of the unchecked downcast is that we can proceed with the variable `square` having type `Square`, despite it having the structure of a `Circle` without the compiler complaining. This implies that when we try to access the field `edgeSize` in line 19, the field will not exist in the object `square` and, hence, we will print `undefined`.

Example 2: Dynamic Property Access

In Figure 1.4 we showcase the TypeScript code corresponding to the key-value map we presented in the JavaScript code in Figure 1.2. The TypeScript code uses classes instead of closures to define `MyMap`. Classes are not specific to TypeScript; they were introduced in ECMAScript 2015 [32]. JaVerT operates on ES5 code and since classes were not available in ES5, we compile them to closures, like in Figure 1.2.

In line 2 of the example, we declare a private field called `contents`. Its type signature, which is `{ [key: string]: number }`, intuitively says that each field in `contents` is denoted by a `string` and that the value of that field is a `number`. The type annotation `[key: string]: number` is referred to as the *index signature* of an object. The index signature can be coupled with other fields, as shown in Figure 1.5; those fields must have a type compatible with the index signature.

```
interface EnhancedMap {
  size: number;
  [key: string]: number;
}

var x: EnhancedMap = {
  size: 2,
  a: 32,
  b: 64
};
```

FIGURE 1.5: Combining an index signature with other fields

Intuitively, we would expect `this.contents.hasOwnProperty` to have type `number` on line 9 due to the index signature. However, *TypeScript assumes prototype safety* and types `this.contents.hasOwnProperty` with the type `(k: string): boolean`, which is the type it has in `Object.prototype`.

The definition of the `get` and `put` methods is identical to that in Figure 1.2. The union type used in the return type of the `get` function is required because not all paths in the function contain an explicit `return` statement; if the key `k` is not in the map, we do not enter the `if` and implicitly return `undefined`. The `private` and `public` identifiers are orthogonal to the discussion about the type system. Nevertheless, we note that they are used only by the TypeScript compiler and have no effect on the code emitted.

Despite the program type checking, a runtime error will be thrown at runtime due to `this.contents.hasOwnProperty` evaluating to a number rather than a function in line 9. The type system cannot check that special identifiers, such as `hasOwnProperty`, are not passed as arguments to the `put` function, thus allowing the shadowing of the function in `Object.prototype` and failing to ensure the prototype safety which it assumes when typing the code.

1.2.3 Examples in JaVerT

The examples presented in JaVerT provide identical code to the JavaScript ones presented in §1.2.1. The code is annotated with JS Logic assertions computed based on the type annotations of the TypeScript code presented in §1.2.2.

Example 1: Flow sensitivity

In Figure 1.6 we present the JavaScript code annotated with JaVerT assertions generated from the TypeScript code in Figure 1.3.

In lines 2-5 we are defining a `Circle` predicate. This predicate corresponds to the `Circle` interface in the TypeScript code. The predicate is formed out of three conjuncts:

1. In line 3, our predicate requires that a `Circle` is a JavaScript object having the prototype set to `Object.prototype`. In order to express that, we are making use of the built-in `JSObjWithProto` predicate;

2. In line 4, we are specifying that an object satisfying the `Circle` predicate must have a data property `radius`, whose value we refer to using the logical variable `#r`; as a rule, all logical variables must begin with the # symbol and are implicitly existentially quantified. `DataProp` is a JaVerT built-in predicate describing an enumerable, writable, and configurable property.

```
 1  /**
 2    @pred Circle(c) :
 3      JSObjWithProto(c, Object.prototype) *
 4      DataProp(c, "radius", #r) *
 5      types(#r : Num);
 6
 7    @pred Square(s) :
 8      JSObjWithProto(s, Object.prototype) *
 9      DataProp(s, "edgeSize", #e) *
10      types(#e : Num);
11  */
12
13  /**
14    @id createShape
15
16    @pre (shape == "circle") * types(sz : Num)
17    @post (ret == #shape) * Circle(#shape)
18
19    @pre (shape == "square") * types(sz : Num)
20    @post (ret == #shape) * Square(#shape)
21  */
22  function createShape(shape, sz) {
23      if (shape === "circle") {
24          return { radius: sz };
25      }
26      else if (shape === "square") {
27          return { edgeSize: sz };
28      }
29  };
30  var shape = createShape("circle", 3);
31  var square = shape;
32  console.log(square.edgeSize);
```

FIGURE 1.6: JavaScript code annotated with JS Logic assertions corresponding to the TypeScript code in Figure 1.3

3. Lastly, in line 5, we require that the value represented by `#r` has type `Num`, meaning that `#r` is a number.

All these conjuncts in the `Circle` predicate describe the way a partial heap containing a `Circle` object should look like. The `Square` predicate (lines 7-10) is similar.

Lines 13-21 represent the specifications for the `createShape` function. The `id` in line 14 is used to disambiguate functions which might have the same name in the code. We provide two distinct sets of pre-/post-conditions, one for the case when `shape` represents the string `"circle"` and the other one for the case when `shape` represents the string `"square"`. The string literal union type used to annotate the `shape` parameter in the TypeScript code in Figure 1.3 provided us with the information required to write two separate sets of specifications. In line 16 we express a pre-condition: `shape` must represent the string `"circle"` and the second argument of the function, `sz`, must be a number. In the post-condition (line 17), we require that the return value, denoted by the logical variable `#shape`, satisfies the `Circle` predicate. The specification for the case when `shape` represents the string `"square"` is almost identical.

Both specifications are satisfied by the implementation and can be verified with JaVerT.

In the call in line 30, we match the first pre-condition of `createShape` and due to the

```
MyMap.prototype.put = function(k, v) {
  if(k !== "hasOwnProperty") {
    this.contents[k] = v;
  }
}
```

FIGURE 1.7: Definition of the put method that ensures prototype safety.

specification of the function, which was verified by JaVerT, we know that the `shape` variable will satisfy the `Circle` predicate. The assignment in line 31 leaves the underlying structure described by the `Circle` predicate unchanged and, hence, when we attempt to access the `edgeSize` property in line 32, we detect that no such property exists and raise an error during verification.

Example 2: Dynamic Property Access

The TypeScript in Figure 1.4 compiles to the JS Logic annotated JavaScript code in Figure 1.8. In the example in Figure 1.6, we began by translating the TypeScript interfaces to JaVerT assertions. Similarly, in lines 1-17, we provide predicates corresponding to the classes we defined in TypeScript. These predicates, one corresponding to instances and one corresponding to the prototype, must ensure prototype safety.

The `MyMapProto` predicate (lines 2-5) describes the prototype of `MyMap`. It contains two properties: `get` and `put`, both functions. The `JSFunctionObject` built-in predicate describes JavaScript function objects. To ensure prototype safety, we require that the prototype of `MyMap` *only* contains the fields `get` and `put` via the `emptyFields` built-in predicate (line 4). This is a stronger requirement than the one in (P2).

Similarly, the `MyMap` predicate requires that:

1. The object `m` has a field `contents`, captured by the (logical) variable `#c`;
2. The object `#c` has an associated index signature[2]. We describe what it means to have an index signature, such as the one in Figure 1.4, with the user-defined predicate `IndexSig`;
3. No properties other than `contents` exist on `m`; this prevents `m` from shadowing any properties that are to be defined in its prototype chain. This condition is stronger than (P1) and clearly ensures prototype safety.
4. The contents, denoted by `#c`, must *not* have the property `hasOwnProperty`.

The `IndexSig` predicate ensures that all the values stored in a certain object are numeric, as required by the type signature in 1.4. To pass verification, it is required that we specify the negative resource in the definition `[recmissing]`.

The specification for the `put` function is the key to ensuring prototype safety. Its post-condition requires that the `this` object satisfies the `MyMap` predicate, which clearly states that the map contents must *not* contain `hasOwnProperty`. However, the function `put` can be called with the parameter `k` having value `"hasOwnProperty"`. Hence, JaVerT is unable to prove the specification. We can pass verification by modifying the definition of `put` to that in Figure 1.7.

The specification of the `get` function requires two different post-conditions. This is because not all the paths throughout the function lead to `return` statement–if a key is not

[2]Index signatures are described when presenting the TypeScript examples in §1.2.2

```
/**
  @pred MyMapProto(mp):
    JSObjWithProto(mp, Object.prototype) * DataProp(mp, "get", #get_loc) *
    DataProp(mp, "put", #put_loc) * emptyFields(mp : -{ "get", "put" }-) *
    JSFunctionObject(#get_loc, "get") * JSFunctionObject(#put_loc, "put");

  @pred MyMap(m, mp):
    JSObjWithProto(m, mp) * DataProp(m, "contents", #c) * IndexSig(#c, #fs) *
    emptyFields(m : -{ "contents" }-) * ((#c, "hasOwnProperty") -> none);

  @pred IndexSig(c, fields):
    [base] fields == [],
    [recexists] (fields == -u- (-{ #f }-, #other)) * DataProp(c, #f, #v) *
                types(#v : Num) * IndexSig(c, #other),
    [recmissing] (fields == -u- (-{ #f }-, #other)) * ((c, #f) -> none) *
                 IndexSig(c, #other);
*/
var MyMap = (function () {
  /**
    @id: constr

    @pre: JSObjWithProto(this, mp) * MyMapProto(mp) * emptyFields(this : -{}-)
    @post: MyMap(this, mp) * MyMapProto(mp)
  */
  function MyMap() { this.contents = {}; }

  /**
    @id: get

    @pre: MyMap(this, mp) * MyMapProto(mp) * types(k : Str)
    @post: MyMap(this, mp) * MyMapProto(mp) * types(ret : Num);
           MyMap(this, mp) * MyMapProto(mp) * types(ret : Undef)
  */
  MyMap.prototype.get = function (k) {
    if (this.contents.hasOwnProperty(k)) { return this.contents[k]; }
  };

  /**
    @id: put

    @pre: MyMap(this, mp) * MyMapProto(mp) * types(k : Str, v : Num)
    @post: MyMap(this, mp) * MyMapProto(mp) * types(ret : Undef)
  */
  MyMap.prototype.put = function (k, v) { this.contents[k] = v; };

  return MyMap;
}());

var myMap = new MyMap();
myMap.put("myKey", 3);
myMap.put("hasOwnProperty", 0);
console.log(myMap.get("myKey"));
```

FIGURE 1.8: JavaScript code annotated with JS Logic assertions corresponding to the TypeScript code in Figure 1.4

present in the map, the function returns `undefined`. This is consistent with the semantics of the types in TypeScript.

JaVerT first verifies the specifications of the defined functions. As the specification of `put` fails, we cannot verify the entire program either. If we replace the definition of the `put` method with that in Figure 1.7, prototype safety is ensured and verification succeeds.

1.3 Contributions

We discussed two approaches to ensuring safety of JavaScript programs. On the one hand, we have TypeScript, which is a superset of JavaScript allowing developers to add type annotations to their code. The type checking process is not sound, as discussed in 1.2.2, but it is very flexible and easy to use from the perspective of the developer. On the other hand, we have JaVerT, which enables programmers to verify functional correctness properties of their code, but requires developers to annotate their code with method specifications and instructions that enable the semi-automatic verifier to carry out its work. This process is cumbersome and is currently justified only in the case of critical JavaScript code.

This project bridges the gap between the flexibility TypeScript provides with the cost of unsoundness and the precision JaVerT offers while asking developers for significant input in the process of verification. The three major contributions of this project are:

1. **A formal translation from TypeScript types to JaVerT assertions**. We interpret types differently than TypeScript in two ways: **(1)** we assume all objects are self-contained, disjoint from all other objects; and **(2)** we impose a stricter definition of classes, which enforces prototype chain structure and guarantees prototype safety.
2. **A soundness proof for the translation**. We prove that, for a TypeScript typing environment Γ [6], all heaps H that satisfy the translation of Γ, also satisfy the typing environment Γ;
3. **A tool that processes easy-to-annotate TypeScript files and compiles them to JavaScript code annotated with JaVerT assertions**. The system offers an expressive interface by supporting a large subset of the TypeScript type annotations. The approach we take is that of semi-automatic specification generation—the generated assertions can be manually perfected by the programmmer.

1.4 Report outline

The present work is structured in five chapters.

Chapter 2 introduces the TypeScript programming language and the JaVerT toolchain. We present the syntax of TypeScript, its sources of unsoundness, and the TypeScript compiler. We proceed by giving a brief introduction to separation logic, after which we introduce JaVerT assertions and the built-in predicates which we use throughout the current work. Finally, we present the safety guarantees offered by JaVerT.

Chapter 3 contains the bulk of the current work. We present the translation of TypeScript type annotations to JaVerT assertions, firstly not taking TypeScript classes into consideration. We then enhance the translation with classes and walk the reader through a practical example. We prove the translation is sound and offer a glimpse of the implementation in the end of the chapter.

In Chapter 4, we evaluate the current work from a theoretical and a practical standpoint. We go in detail regarding known limitations and compare our tool with other works. We briefly mentioned the lessons learnt while developing the project.

Chapter 5 presents the future work that can be carried out in the current space. We present several potential improvements for our current tool as well as brining forward a discussion regarding the relation between separation logic and types.

Chapter 2

Background

As it was mentioned in introduction, we aim to combine the advantages of two distinct approaches of ensuring safety in JavaScript: retrofitting a type system on top of JavaScript and verifying functional correctness via the use of assertions. The examples presented in §1.2 showcased what the strengths and weaknesses of these approaches are, but did not go into detail regarding their inner workings and the guarantees they provide.

We present the TypeScript language and the JaVerT toolchain, focusing on the syntax and the semantics of TypeScript types and JaVerT assertions respectively. We discuss their safety guarantees and their drawbacks briefly to get a clear image of what is the underlying foundation this project is built upon.

2.1 TypeScript

TypeScript is an extension of JavaScript offering interfaces and a gradual type system. It is designed to facilitate easy adoption for JavaScript developers. Its users enjoy a streamlined IDE experience and a certain degree of type safety — the TypeScript type system is not sound by design so errors can still occur despite the program type checking. Safe alternatives to the TypeScript type system exist: in [38] a system is developed involving more thorough static checking of the code coupled with run-time instrumentation to ensure type safety.

Syntactically, TypeScript is a superset of JavaScript — every valid JavaScript program is a valid TypeScript program. The TypeScript compiler processes TypeScript code and emits JavaScript. The compilation is more complicated than simply removing type annotations: the TypeScript compiler gives the user the option to specify the target JavaScript standard; additionally, some features are often available in TypeScript before they are available in JavaScript (e.g. classes, async/await keywords).

We present TypeScript, following [6]. We discuss the specifics of the type system, covering the syntax of types and some of the novel types TypeScript provides such as intersection types and union types [3]. Sources of unsoundness are presented and illustrated with examples. We present the notation that we use throughout the project and introduce the TypeScript compiler.

2.1.1 Syntax of TypeScript

We describe the syntax of the subset of TypeScript covered in this project. It is very similar to the syntax of [6]. The fragment of TypeScript which we choose not to tackle is discussed in §4.3.

$$
\begin{array}{rccl}
\text{Identifiers} & \texttt{x} & \in & \mathcal{ID} \\
\text{Literals} & \texttt{l} & \in & \mathcal{R} \cup \mathcal{S} \cup \{\ \texttt{true},\ \texttt{false}\ \} \\
\text{Binary operators} & \oplus & \in & \{+, -, *, \cdots\} \\
\text{Expressions} & \texttt{e} & ::= & \texttt{x} \mid \texttt{l} \mid \texttt{undefined} \mid \texttt{null} \mid \texttt{e}_1 \oplus \texttt{e}_2 \\
 & & \mid & \{\ \texttt{n}_i : \texttt{e}_i|_{i=0}^{n}\ \} \mid \texttt{e.n} \mid \texttt{e}_1[\texttt{e}_2] \mid \texttt{e}_1 = \texttt{e}_2 \\
 & & \mid & \texttt{e}(\texttt{e}_i|_{i=0}^{n}) \mid \texttt{new e}(\texttt{e}_i|_{i=0}^{n}) \mid \texttt{e as } \tau \mid (\texttt{e}) \\
 & & \mid & \texttt{function } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\texttt{s}}\ \} \\
\text{Statements} & \texttt{s} & ::= & \texttt{e} \mid \texttt{if (e) \{ } \overline{\texttt{s}} \texttt{ \} else \{ } \overline{\texttt{s}} \texttt{ \}} \\
 & & \mid & \texttt{while (e) \{ } \overline{\texttt{s}} \texttt{ \}} \mid \texttt{return;} \\
 & & \mid & \texttt{return e;} \mid \texttt{var x: } \tau\texttt{;} \\
 & & \mid & \texttt{var x: } \tau \texttt{ = e;} \mid \texttt{F} \\
\text{Named function declarations} & \texttt{F} & ::= & \texttt{function x}(\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\texttt{s}}\ \}
\end{array}
$$

FIGURE 2.1: The syntax of TypeScript expressions and statements

In the syntax presented we use the notation $\overline{\texttt{A}}$ to denote that the terminal or non-terminal A may appear zero or more times, $\underline{\texttt{A}}$ to denote that the terminal or non-terminal A is optional (i.e. may appear zero times or once) and $\texttt{A}^+$ to denote that the terminal or non-terminal A appears one or more times.

The syntax of programs. TypeScript expressions and statements, presented in Figure 2.1, are largely inheritted from JavaScript. A major difference in between the subset of TypeScript we support and the calculus presented in [6] is that we require type annotations for function parameters, function return values and variable declarations, while in [6] type annotations are optional for all these cases. This requirement is largely because we are dependent on type annotations to produce JaVerT assertions and creating an inference mechanism is beyond the scope of the project.

We define a literal to be either a number literal $\texttt{r} \in \mathcal{R}$, where $\mathcal{R}$ is the set of all real numbers expressible in JavaScript (e.g. $\texttt{r} = 3.14$), a string literal $\texttt{str} \in \mathcal{S}$ (e.g. $\texttt{str} =$ `"abc"`), or a boolean literal $\texttt{b} \in \{\ \texttt{true},\ \texttt{false}\ \}$. The coexistence of `undefined` and `null` can be confusing; the difference between them is mostly semantic—`null` is used to denote a non-existence reference, while `undefined` is the value of an uninitialised variable. The cases for expressions are mostly straightforward; the novel ones are $\texttt{e as } \tau$, which represents a cast of expression e to type τ and $\texttt{function } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\texttt{s}}\ \}$ which highlights that functions are first class citizens in JavaScript and implicitly in TypeScript. Statements are standard; we ignore some of the regularly used constructs such as `for` loops from our syntax for brevity as they can easily be expressed using a `while` loop. `const`/`let` declarations are omitted as their effect is mainly on the scoping rules, a subject orthogonal to that of the current work.

The syntax of type annotations. We present the syntax of TypeScript type annotations in Figure 2.2. A type is one of the following: (1) the special type `any`, presented in the examples in §2.1.2, (2) a primitive type, ρ, (3) a type name, T, denoting a user-defined type, (4) an object type literal, O.

Primitive types are standard and can be found in many other languages. It is worth noting that both `void` and `undefined` require an identifier to represent the value `undefined`; as a convention `void` is used to denote the return type of a function with no returned value, while `undefined` is associated with variables and it signifies that their value is `undefined`. The type `null` signifies that a variable holds the value `null`.

Type names are associated with user-defined types: interfaces or classes. We split the set of type names $\mathcal{T}$ in two disjunct subsets: interface names $\mathcal{I}$ and class names $\mathcal{C}$.

Type names	`T`	$\in$	$\mathcal{T}$
Interface names	`I`	$\in$	$\mathcal{I}$
Field names	`n`	$\in$	$\mathcal{F}$
String literals	`str`	$\in$	$\mathcal{S}$
Primitive types	ρ	::=	`number` | `boolean` | `string` | `void` | `str`
		|	`undefined` | `null`
Types	τ	::=	`any` | ρ | `T` | `O` | $(\tau_1 \mid \tau_2)$
Object type literals	`O`	::=	$\{\ \mathtt{n}_i : \tau_i\|_{i=0}^{n},\ \mathtt{n}'_j\ ?: \tau'_j\|_{j=0}^{m}\ \}$
		|	$\{\ \mathtt{n}_i : \tau_i\|_{i=0}^{n},\ \mathtt{n}'_j\ ?: \tau'_j\|_{j=0}^{m},\ [\mathtt{n} : \mathtt{string}] : \tau\ \}$
		|	$\{\ (\mathtt{x}_i : \tau_i\|_{i=0}^{n}) : \tau_r,\ \mathtt{n}'_j : \tau'_j\|_{j=0}^{m},\ \mathtt{n}''_k\ ?: \tau''_k\|_{k=0}^{p}\ \}$
		|	$\{\ \mathtt{new}(\mathtt{x}_i : \tau_i\|_{i=0}^{n}) : \tau_r,\ \mathtt{n}'_j : \tau'_j\|_{j=0}^{m},\ \mathtt{n}''_k\ ?: \tau''_k\|_{k=0}^{p}\ \}$
Interface decl.	`ID`	::=	`interface I O`
		|	`interface I extends` $\mathtt{T}^{+}$ `O`

FIGURE 2.2: The syntax of TypeScript type annotations

Union types $\tau_1 \mid \tau_2$ indicate that a certain object can either be of type τ_1 or type τ_2.

Object types `O`, are specific for a structural type system: they specify what fields should be present on an object. The following members can be present on an object:

A regular field `n`: τ, indicating field `n` has type τ

An optional field `n ?`: τ indicates that the field `n` may or may not be present on an object of the current type, and if it is present it has type τ.

A call signature $(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r$, indicating that the current object represents a function with n arguments returning a value of type τ_r. The type of the i^{th} argument is τ_i;

An index signature `[n: string]`: τ indicating that the current object can be indexed with any `string` and if that field exists, the result is an object of type τ;

A constructor signature $\mathtt{new}(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r$ meaning the current object represents a constructor (i.e. a function that should be called using the `new` keyword). The constructor call requires n arguments, similar to the call signature and creates an object of type τ_r.

Different approach to object literal types: Our definition of an object literal is different from that presented in [6]. While they allow multiple call signatures, index signatures and constructor signatures in an object literal type, we restrict each one of these declarations to at most one. This design decision is based on the fact that while multiple declarations are possible, they need to be satisfied by a single implementation, which must weaken its type to fit all the defined overloads. Hence, we find that a single definition for all the aforementioned members is easier to reason about and enables us to provide clearer assertions. Similarly, we distinguish amongst object type literals with associated call signatures, constructor signatures or none of these. An object literal having both a call signature and a constructor signature would require an object `o` satisfying the type to have constructor behaviour (i.e. create a new object when called using the `new` keyword), as well as regular function behaviour (i.e. correct behaviour when `this` is bound to a previously existent object). Hence, we chose to make it impossible for an object type literal to contain both these declarations.

Interfaces allow us to specify the shape of an object and associate a name with it; they are used for type checking only—no JavaScript is emitted for the declaration of an interface. Interfaces are the named variants of object type literals `O`: a declaration `interface`

$$
\begin{array}{rcll}
\text{Class names} & \mathtt{C} & \in & \mathcal{C} \\
\text{Access specifiers} & \kappa & \in & \{\ \mathtt{public},\ \mathtt{private},\ \mathtt{protected}\ \} \\
\text{Class declarations} & \mathtt{C} & ::= & \mathtt{class\ C\ \{\ CD,\ \overline{M}\ \}} \\
 & & | & \mathtt{class\ C\ extends\ C_e\ \{\ CD,\ \overline{M}\ \}} \\
 & & | & \mathtt{class\ C\ implements\ T^{+}\ \{\ CD,\ \overline{M}\ \}} \\
 & & | & \mathtt{class\ C\ extends\ C_e\ implements\ T^{+}\ \{\ CD,\ \overline{M}\ \}} \\
\text{Constr. definition} & \mathtt{CD} & ::= & \underline{\kappa}\ \mathtt{constructor}(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\mathtt{s}}\ \} \\
\text{Members} & \mathtt{M} & ::= & \underline{\kappa}\ \mathtt{n}:\ \tau; \\
 & & | & \underline{\kappa}\ \mathtt{n}:\ \tau\ \mathtt{=}\ \mathtt{e}; \\
 & & | & \underline{\kappa}\ \mathtt{x}(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\mathtt{s}}\ \}
\end{array}
$$

FIGURE 2.3: The syntax of TypeScript class declarations

`I O` associates object type literal `O` with the name `I`. Interfaces can extend zero or more base types. All fields introduced by `I` or any of its base types must be present on an object of type `I`. No access modifiers are allowed so all members declared are considered public and are inherited from the base types. Circular dependencies are forbidden. Interfaces can also extend classes. This is discussed after classes are formally introduced.

The syntax of classes. We introduce TypeScript classes in Figure 2.3. Classes are syntactic sugar over JavaScript's prototype-based inheritance; they do not introduce a new object oriented inheritance model to JavaScript. They can appear in both declarations and expressions; without loss of generality we focus on `Class` declarations.

A class can `extend` another class. Saying a `class A` extends a `class B` makes `B` the immediate parent on the prototype chain of `A`. A class can `implement` multiple interfaces or classes. If `class A` implements `interface I`, then all objects of type `A` must adhere to the shape indicated by `I`. This implies that all methods specified by `interface I` are associated with an implementation in `class A` and all other fields named are present on every instance of `A`.

As mentioned, a `class A` can `implement` another `class B` and an `interface I` can extend a `class B`. In this context `class B` is treated as an interface: the method implementations are stripped away and the definitions of the class' members treated as if they were part of an interface.[1]

2.1.2 Sources of unsoundness

We present the three sources of unsoundness discussed in [6]. Dynamic property accesing is an additional source of unsoundness, which is not covered here as we illustrated it with the code in Figure 1.4.

1. **Unchecked downcasts.** Type erasure is one of the characteristics of TypeScript. This design decision makes it impossible to benefit from run-time type checks whenever using implicit or explicit downcasts. This causes a hole in the type system as illustrated in Figure 1.3 in §1.2.2.
2. **Unchecked gradual typing.** By design, TypeScript wants JavaScript programmers to encounter as few issues as possible in migrating to TypeScript and decided to opt for a gradual type system to achieve this goal. Gradual typing is achieved by

[1]Implementing other classes does not work when access specifiers other than `public` are used in the *parent* class. A GitHub issue tracking the subject is available here: https://github.com/Microsoft/TypeScript/issues/471 — accessed 23 Feb 2018

```
type Vegetable = "carrot" | "potato";
interface Person {
  eat: { (food: any): void };
}
interface Vegetarian {
  eat: { (food: Vegetable): void };
}
var veg: Vegetarian = {
  eat: function(food: Vegetable): void {
    console.log("Eating vegetable " + food);
  }
}
var pers: Person = veg;
pers.eat("beef"); // "Eating vegetable beef"
```

(A) Type checking program illustrating unsoundness caused by the use of unchecked covariance on object properties

```
interface MathLib {
  pi: number;
  isPrime: {
    (x: number): boolean
  };
  nextPrime: {
    (x: number): number
  };
  floor: {
    (x: number): number
  }
}

var num: number = 3;
var x: any = num;
var obj: MathLib = x;
```

(B) Type checking program illustrating unsoundness caused by use of any

FIGURE 2.4: Examples of unsoundness in TypeScript.

using the special type `any`. The assignment compatibility rules involving `any` are very liberal: anything can be assigned to `any` and `any` can be assigned to anything. We present in Figure 2.4b a code snippet in which using a temporary variable of type `any` enables us to bypass the type system and assign a `number` to a variable that is expected to have the type `MathLib`.

3. **Unchecked covariance.** To have a sound type system, the types of object fields need to be invariant for two objects to be assignable to each other. TypeScript takes a different approach and allows covariance on object property types, as illustrated by the code in Figure 2.4a. TypeScript allows us to assign expressions with type Vegetarian to variables of type Person because the type of the field `eat` in an object of type `Vegetarian` is a subtype of the field `eat` in type `Person`. The unsoundness is apparent, since after assigning the `Vegetarian veg` to `pers` of type `Person`, we can make `veg` eat `"beef"`.

2.1.3 Useful notation

We define a typing environment Γ as a mapping from identifiers to TypeScript types. We write $\cdot$ to denote the empty type environment and $\Gamma, \mathtt{x} : \tau$ to denote the extension of the type environment Γ with the mapping of identifier x to type τ. Environment extension is only defined if $\mathtt{x} \notin dom(\Gamma)$. In the following section we use the above notation to analyse the mapping $\mathtt{x} : \tau$ for every $\mathtt{x} \in dom(\Gamma)$; hence, we write $\Gamma = \Gamma', (\mathtt{x} : \tau)$ to obtain the mapping for identifier $\mathtt{x} \in dom(\Gamma)$.

2.1.4 The TypeScript compiler

The official TypeScript compiler, `tsc`, provides an API that is instrumental in our efforts of translating type annotations to JaVerT assertions. Alongside the compiler API, `tsc` performs several useful checks, such as ensuring that the program is valid syntactically, and translates it to JavaScript.

```
var n1: number = undefined; // <- Fails to compile. 'undefined' cannot be
    assigned to type 'number'.
var n2: number | undefined = undefined // <- Compiles since we explicitly
    mentioned the value can be undefined.
var n3: number | undefined = 3 // <- Compiles since 3 is a number
var n4: number | undefined = null // Fails to compile. 'null' cannot be
    assigned to type 'number'.
```

FIGURE 2.5: Example of TypeScript behaviour when the `strictNullChecks` flag is activated.

Compiler options

Target. JaVerT works on JavaScript ES5 [13] programs annotated with assertions. In this work we use features of TypeScript that require translation to become valid ES5 code. The most significant is the `class` feature which is only introduced in JavaScript from ES2015 [14]. We can require `tsc` to output valid ES5 via the `target` compiler option.

Strict null checks. Every variable in TypeScript can by default, regardless of its type, hold the falsy values `undefined` or `null`. The `tsc` compiler prohibits this behaviour when the `strictNullChecks` compiler option is activated [27], and requires types to explicitly state that a variable can hold `undefined` or `null`. An example is provided in Figure 2.5. We aim to provide safe behaviour in TypeScript code and opt for the behaviour imposed by running `tsc` with `strictNullChecks` enabled.

The TypeScript compiler API

The TypeScript compiler's functionality can be accessed by client TypeScript code via the API provided. This enables software such as ours to build additional functionality around the TypeScript compiler.

Besides offering developers the possibility to alter the compile options or load additional files, the TypeScript compiler API gives programmers access to the generated TypeScript abstract syntax tree (AST). The design employed is well-established, visitors must be used to walk the AST and potentially modify it.

2.1.5 Translating a TypeScript `Class` to ES5

TypeScript classes are the only entities in our syntax which do not have a close equivalent in ES5. Their translation to ES is fortuantely handled by `tsc` and hence does not impose any burden on our software. In Figure 2.6 we illustrate a simple instance of class extension. We imagine a number of animals placed on a 1-dimensional axis. Upon creation, an animal is at position 0 and can move to the right (positive distance) or to the left (negative distance).

`Animal`, the base class, provides a single private member, `position`, the initial position of every `Animal` and a method, `walk`, which enables the animal to move either direction;

`Cat` inherits from `Animal`; a `Cat` has a name and can `meow`. As is the case in most object-oriented programming languages, the `Cat` constructor must include a call to `super` in order to populate the fields defined in the superclass.

Since JaVerT operates on ES5, we make the transition to the world of JavaScript pre-classes. The transpiled code is featured in Figure 2.7.

```
1  class Animal {
2     private position: number = 0;
3     walk(distance: number) {
4        this.position += distance;
5     }
6  }
7
8  class Cat extends Animal {
9     private name: string;
10    constructor(name: string) {
11       super();
12       this.name = name;
13    }
14    meow() {
15       console.log("Meow!! I'm " + name + "!");
16    }
17 }
```

FIGURE 2.6: TypeScript code showcasing classes and class extension.

Class translation with no inheritance. The translation for the `Animal` class is much simpler than that of the `Cat` class, as no inheritance is involved. Lines 1-6 in Figure 2.6 translate to lines 1-8 in the JavaScript code presented in 2.7. We analyse the JavaScript code line-by-line. On line 1 we have a comment `/** @class */` that indicates the expression following is corresponding to a class. The creation of the class constructor, defined in between lines 2-4, is wrapped in a function. The instance methods are then added to the prototype of the constructor function and finally, the constructor is returned in line 7 and stored in a variable, `Animal`. The `Animal` constructor can now be used together with the `new` keyword to create objects behaving as an instance of the `Animal` class would in an object oriented language. We emphasize that this behaviour is achieved via prototypal inheritance in JavaScript.

Class translation when the class inherits from a parent class. The definition of the `Cat` class, lines 8-17 in Figure 2.6, gets translated to lines 9-20 in Figure 2.7. The translation uses an automatically generated function, `__extends`, which sets the `__proto__` field of the inheriting objects to their corresponding parent in the prototype chain. This function is common for all instances of inheritance and is fully handled by `tsc`. Once the purpose of the `__extends` function is clear, the definition of the `Cat` class can be easily parsed in the same manner as that of the `Animal` class. The main difference is that the `_super` argument, taken by the function on line 9, is used to set the parent in the prototype chain (line 10) and populate the required fields of the `this` object (line 12).

2.2 JaVerT

We introduce separation logic as it constitutes the foundation upon which the JaVerT assertions are built. A presentation of JaVerT assertions, covering some of the built-in abstractions follows. We then present the structure of the JaVerT toolchain and discuss its safety guarantees. Most of the information is adapted from [17].

2.2.1 Separation logic

Separation logic [35] is an extension of Hoare logic [21] that permits reasoning about low-level imperative programs that use shared mutable data structures.

```
var Animal = /** @class */ (function () {
    function Animal() {
        this.position = 0;
    }  Animal.prototype.walk = function (distance) {
        this.position += distance;
    };
    return Animal;
}());
var Cat = /** @class */ (function (_super) {
    __extends(Cat, _super);
    function Cat(name) {
        var _this = _super.call(this) || this;
        _this.name = name;
        return _this;
    }
    Cat.prototype.meow = function () {
        console.log("Meow!! I'm " + name + "!");
    };
    return Cat;
}(Animal));
```

FIGURE 2.7: The JavaScript emitted after the compilation of the TypeScript code in 2.6

Hoare introduces a new notation: $P\{C\}Q$ to be interpreted as "If the assertion P is true before the initialisation of a program C, then the assertion Q will be true on its completion." [21] His system includes a set of rules for reasoning rigorously about the correctness of imperative programs that alter the variable store. However, Hoare logic is not suitable for reasoning about imperative programs that alter the heap—it does not scale due to the need to explicitly reason about overlaps.

Separation logic [35] extends the assertion language with operators for describing memory heaps. It introduces two main operators: $\mapsto$ and $*$. The assertion $l \mapsto v$, read "l maps to v" describes the heap consisting of **only one** memory cell, at address l, containing value v. The assertion $P * Q$, where $*$ is the separating conjunction, describes all the heaps that can be split into two **disjoint** heaps, one of them satisfying assertion P and the other one satisfying assertion Q. The assertion emp describes the empty heap.

For example, the assertion $10 \mapsto 1 * 11 \mapsto 2$, describes a heap containing two memory cells, at addresses 10 and 11, storing values 1 and 2, respectively. To relate this to the semantics of the separating conjunction, we can split the heap described into two distinct heaps, one containing one memory cell at address 10 holding value 1 and another heap formed of one memory cell at address 11 holding value 2.

All the usual logic operators can be used in conjunction with the separation logic ones.

2.2.2 JaVerT assertions

To specify JavaScript programs, JaVerT provides an assertion language capable of capturing key JavaScript heap structures such as: property descriptors, prototype chains, the variable store emulated via scope chains, and function closures.

JavaScript Memory Model

JS locations : $l \in \mathcal{L}$ JS variables : $x \in \mathcal{X}_{\text{JS}}$ JS heap values : $\omega \in \mathcal{V}^h_{\text{JS}} ::= v \mid \overline{v} \mid fid$

JS values: $v \in \mathcal{V}_{\text{JS}} ::= n \mid b \mid m \mid \text{undefined} \mid \text{null} \mid l$ JS heaps : $h \in \mathcal{H}_{\text{JS}} : \mathcal{L} \times \mathcal{X}_{\text{JS}} \rightharpoonup \mathcal{V}^h_{\text{JS}}$

A JavaScript heap, $h \in \mathcal{H}_{\text{JS}}$, is a partial function mapping pairs of object locations and property names to JS heap values.

Object locations are taken from a set of locations $\mathcal{L}$. Property names and JS program variables are taken from a set of strings $\mathcal{X}_{\text{JS}}$.

JS values contain: numbers, n; booleans, b; strings, m; the special JavaScript values `undefined` and `null`; and object locations, l.

JS heap values, $\omega \in \mathcal{V}^h_{\text{JS}}$, contain: JS values, $v \in \mathcal{V}_{\text{JS}}$; lists of JS values, $\overline{v}$; and function identifiers, $\mathit{fid} \in \mathcal{F}\mathit{id}$.

Function identifiers, fid, are associated with syntactic functions in the JavaScript code and are used to represent function bodies in the heap uniquely. [...] The ECMAScript standard does not prescribe how function bodies should be represented and our choice closely connects the JavaScript and JSIL heap models.

Notation. Given a heap h, we denote a heap cell by $(l, x) \mapsto v$ when $h(l, x) = v$, disjoint heap union by $h_1 \uplus h_2$, heap lookup by $h(l, x)$, and the empty heap by emp.[2]

JS Logic Assertions

$$V \in \mathcal{V}^L_{\text{JS}} ::= \omega \mid \omega_{\texttt{set}} \mid \varnothing \qquad E \in \mathcal{E}^L_{\text{JS}} ::= V \mid x \mid \mathsf{x} \mid \ominus E \mid E \oplus E \mid \texttt{sc} \mid \mathsf{this}$$

$$\tau \in \texttt{Types} ::= \texttt{Num} \mid \texttt{Bool} \mid \texttt{Str} \mid \texttt{Undef} \mid \texttt{Null} \mid \texttt{Obj} \mid \texttt{List} \mid \texttt{Set} \mid \texttt{Type}$$

$$P, Q \in \mathcal{AS}_{\text{JS}} ::= \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \leq E \mid P \wedge Q \mid \neg P \mid P * Q \mid \exists \mathsf{x}.P \mid \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E \mid E) \mid \mathsf{types}(X_i : \tau_i|_{i=1}^{n})$$

JS logical values, $V \in \mathcal{V}^L_{\text{JS}}$, contain: JS heap values, ω; sets of JavaScript heap values, $\omega_{\texttt{set}}$; and the special value $\varnothing$, read *none*, used to denote the absence of a property in an object.

JS logical expressions, $E \in \mathcal{E}^L_{\text{JS}}$, contain: logical values, V; JS program variables, x; JS logical variables, x; unary and binary operators, $\ominus$ and $\oplus$ respectively; and the special expressions, `sc` and this, referring respectively to the current scope chain and the current this object.

JS Logic assertions are constructed from: basic boolean constants, comparisons, and connectives; the separating conjunction; existential quantification; and assertions for describing heaps and declaring typing information. The emp assertion describes an empty heap. The cell assertion, $(E_1, E_2) \mapsto E_3$, describes an object at the location denoted by E_1 with a property denoted by E_2 that has the value denoted by E_3. The assertion $\mathsf{emptyFields}(E_1 \mid E_2)$ states that the object at the location denoted by E_1 has no properties other than possibly those included in the set denoted by E_2. The assertion $\mathsf{types}(X_i : \tau_i|_{i=1}^{n})$ states that variable X_i has type τ_i for $0 \leq i \leq n$, where X_i is either a program or a logical variable and τ ranges over JavaScript types, $\tau \in \texttt{Types}$. We say that an assertion is *spatial* if it contains cell assertions or ∣emptyFields∣ assertions. Otherwise, it is *pure*.[3]

2.2.3 JS Logic built-in predicates

For JS Logic assertions to be as concise as possible, a number of built-in predicates are provided. We illustrate those that are relevant for the current work; the complete list together with a thorough description is available in [17]:

[2]Description of JavaScript memory model adapted from [17].

[3]Description of JS Logic assertions, as presented in [17].

1. `JSObjectGen(o, p, c, e)` describes an object `o`, with prototype `p`, having the internal `class` attribute value `c` and the `extensible` internal attribute set to `e`;
2. `JSObjectWithProto(o, p)` is a special instance of `JSObjectGen`, where the `class` attribute is set to `"Object"` and `extensible` is set to `true`.
3. `JSFunctionObject(o, fid, sc, len, p)` describes the function object `o` associated with the function body indicated by `fid`. The `scope` internal property has value given by `sc`; there are `len` parameters expected and the `prototype` field has value `p`.
4. `JSFunctionObjectStrong(o, fid, sc, len, p)` is a variant of `JSFunctionObject` which does not allow any user-defined properties on the function object `o`.
5. `DataProp(o, p, v)` states that the property `p` of object `o` holds a data descriptor with value `v` and all other attributes set to `true`. A more general predicate, `DataPropGen`, is available, but we do not make use of it.
6. `DescVal(desc, v)` states that the data descriptor `desc` has the value attribute `v`.
7. `Pi(o, p, d, lo, lc)` states that the property `p` has value `d` in the prototype chain of `o`. The two additional parameters, `lo` and `lc`, denote lists that capture the locationns and classes of the objects in the prototype chain up to and including the object in which `p` is found, or of all objects if the property is not found.
8. `Scope(x : v, sc, fid)` states that the variable `x` has value `v` in the scope chain denoted by `sc` of the function literal with identifier `fid`. It is common to use `Scope(x : v)` to state that in the current scope chain, in the current function, variable `x` has value `v`.
9. `EmptyFields(o | F)` states that the object at the location denoted by `o` has no properties other than possibly those in the set denoted by F.
10. `ObjectPrototype()` decribes the resource of the `Object.prototype` object.
11. `FunctionPrototype()` decribes the resource of the `Function.prototype` object.
12. `GlobalObject()` describes the resource of the global object.
13. `GlobalVar(p, v)` states that in the global object, property p has value v.

Parameters of predicates can be left unspecified, by using _ in their place.

2.2.4 Toolchain structure and safety guarantees

Figure 2.8 presents the infrastructure of JaVerT, as presented in [17]. We describe the individual components to formulate a safety guarantee.

The first stage involves compiling the JavaScript annotated with JS Logic assertions to JSIL, a simple intermediate goto language capturing the dynamism of JavaScript, annotated with JSIL logic assertion. Thus, instead of reasoning about complex JavaScript code, JaVerT reasons about much simpler JSIL statements. The compiler carrying out the work, JS-2-JSIL is meant to be line-by-line close to the ECMAScript standard, without simplifying the behaviour in any way. In addition to this, JSIL subsumes the heap model of JavaScript so their assertions coincide. The JS-2-JSIL compiler is tested against the ECMAScript Test262 suite, while the JS-2-JSIL logic translator is proven correct.

The JSIL code, annotated with JSIL assertions is checked by a semi-automatic verifier, JSIL Verify. JSIL Verify comprises of a symbolic execution engine and entailment engine,

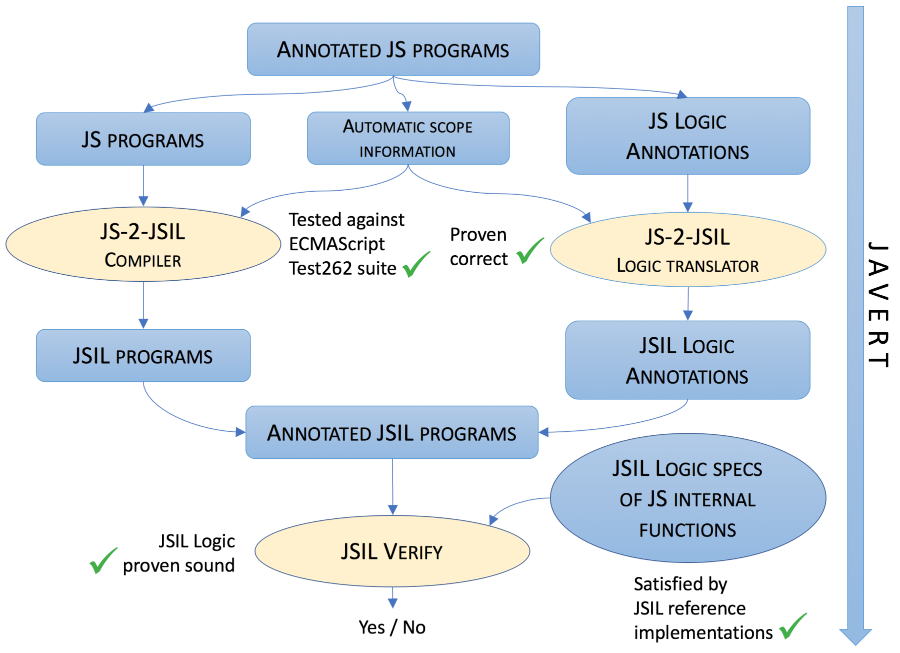
ANNOTATED JS PROGRAMS
JS PROGRAMS
AUTOMATIC SCOPE INFORMATION
JS LOGIC ANNOTATIONS
JS-2-JSIL COMPILER
Tested against ECMAScript Test262 suite ✓
Proven correct ✓
JS-2-JSIL LOGIC TRANSLATOR
JSIL PROGRAMS
JSIL LOGIC ANNOTATIONS
ANNOTATED JSIL PROGRAMS
JSIL LOGIC SPECS OF JS INTERNAL FUNCTIONS
✓ JSIL Logic proven sound
JSIL VERIFY
Yes / No
Satisfied by JSIL reference implementations ✓
JAVERT

FIGURE 2.8: JaVerT infrastructure [17]

which uses Z3 SMT solver [11] to discharge assertions in first-order logic with equality and arithmetic.

JaVerT contains axiomatic specifications for the JavaScript internal functions in JSIL Logic and providing reference implementations in JSIL. The reference implementations are line-by-line close to the standard and proven correct with respect to the axiomatic specifications using JSIL Verify.

In terms of correctness, JS-2-JSIL, has been thoroughly tested: it passed the 8797 tests in the official ECMAScript test suite. JSIL Logic is proven sound with respect to its operational semantics. Since JSIL is designed so that the JSIL heap model subsumes the JavaScript heap model, the correctness of the logic translator is straightforward. A proof showing that JS-2-JSIL is logic-preserving and that JSIL verification lifts to JavaScript verification are presented in [17]. "JSIL Verify is validated by verifying that the reference implementations of the internal functions are correct with respect to their axiomatic specifications, and by verifying compiled JavaScript programs. The specifications of the internal functions are validated by verifying that they are satisfied by their well-tested corresponding JSIL reference implementations."[4]

[4]As specified in [17]

Chapter 3

Translating TypeScript to JaVerT

The translation process consists of two stages. First, we process TypeScript code and emit JavaScript ES5 code that JaVerT can analyse; this is taken care of by the official TypeScript compiler, `tsc` [30]. Second, we translate the type annotations present in the TypeScript code to JaVerT annotations; this translation is presented in the rest of this section. Once both of these translations are done, we combine the outputs and hand them over to JaVerT for the verification step.

Notation. We write: (1) $\mathcal{G}$ for the set of all possible typing environments Γ; (2) $\mathcal{A}$ for the set of all possible JaVerT assertions; (3) $\mathcal{T}$ for the set of all types; (4) $\mathcal{C}$ for the set of all class names; (5) $\mathcal{F}$ for the set of all functions in the program; (6) $\mathcal{N}_{\mathcal{C}}$ for the set of all methods in all classes; (7) $\mathcal{N}_{\mathtt{C}}$ for the set of all methods in class `C`; (8) `m` to iterate over methods and $\mathtt{m}_{\mathtt{C}}$ to indicate that the method `m` is a method of class `C`; (9) $\mathtt{constr}_{\mathtt{C}}$ to denote the constructor of class `C`.

We define five compilers to translate TypeScript type annotations to JaVerT assertions:

The environment compiler, $\mathscr{C}_{TE} : \mathcal{G} \to \mathcal{A}$, takes a typing environment, Γ, and generates a JaVerT assertion describing a heap in which every variable in the domain of Γ has the type indicated by Γ.

The type compiler, $\mathscr{C}_{T} : (\mathcal{E}^{L}_{\mathtt{JS}} \times \mathcal{T}) \to \mathcal{A}$, takes a logical expression and a TypeScript type and generates the JaVerT assertion describing the heap in which the value associated with the logical expression has the given type.

The function spec compiler, $\mathscr{C}^{F}_{spec} : \mathcal{F} \to (\mathcal{A} \times \mathcal{A})$, takes a function `f` and generates its corresponding pre- and post-condition.

The method spec compiler, $\mathscr{C}^{M}_{spec} : \mathcal{N}_{\mathcal{C}} \to (\mathcal{A} \times \mathcal{A})$, generates specifications for class methods.

The constructor spec compiler, $\mathscr{C}^{C}_{spec} : \mathcal{C} \to (\mathcal{A} \times \mathcal{A})$, generates specifications for the constructor of any class $\mathtt{C} \in \mathcal{C}$. To simplify our reasoning when defining $\mathscr{C}^{C}_{spec}$, we require classes to explicitly define a constructor (§2.1.1); if this would not be the case, we could create an empty constructor as part of the pre-processing stage.

The disjointness assumption. In §2.2.1, we introduced the separating conjunction operator, $*$. The assertion $P * Q$ describes all the heaps that can be split into two disjoint heaps, one of them satisfying assertion P and the other one satisfying assertion Q. This separation is a very strong assumption; aliasing is allowed in JavaScript and TypeScript code and the TypeScript typing environment Γ does not provide any information regarding aliasing. Separation logic is not a suitable tool for modelling potential overlaps. We design the analysis assuming that all our variables refer to self-contained objects, disjoint from all other objects defined.

3.1 The translation without classes

In the table below, we define the $\mathscr{C}_T$ compiler for all the supported TypeScript type annotations, except for classes.

Translation of TypeScript type annotations to JaVerT assertions, without `class` types

Type-Void
$\mathscr{C}_T(E, \mathtt{void}) \triangleq \mathsf{types}(E : \mathrm{Undef})$

Type-Undef
$\mathscr{C}_T(E, \mathtt{undefined}) \triangleq \mathsf{types}(E : \mathrm{Undef})$

Type-Null
$\mathscr{C}_T(E, \mathtt{null}) \triangleq \mathsf{types}(E : \mathrm{Null})$

Type-Number
$\mathscr{C}_T(E, \mathtt{number}) \triangleq \mathsf{types}(E : \mathrm{Num})$

Type-String
$\mathscr{C}_T(E, \mathtt{string}) \triangleq \mathsf{types}(E : \mathrm{Str})$

Type-StringLiteral
$\mathscr{C}_T(E, \mathtt{str}) \triangleq (E == \mathtt{str})$

Type-Bool
$\mathscr{C}_T(E, \mathtt{boolean}) \triangleq \mathsf{types}(E : \mathrm{Bool})$

Type-Any
$\mathscr{C}_T(E, \mathtt{any}) \triangleq \mathsf{emp}$

Type-Union
$\mathscr{C}_T(E, \tau_1 | \tau_2) \triangleq \mathscr{C}_T(E, \tau_1) \vee \mathscr{C}_T(E, \tau_2)$

Mandatory fields compiler
$\mathscr{C}_{OF}(E, \mathtt{n}_i : \tau_i|_{i=0}^{n}) \triangleq \circledast_{i=0}^{n} \mathtt{Pi}(E, \mathtt{n}_i, \#desc, \_, \_) * \mathtt{DescVal}(\#desc, \#v) * \mathscr{C}_T(\#v, \tau_i)$

Optional fields compiler
$\mathscr{C}_{OF}(E, \mathtt{n}_i\ \mathtt{?} : \tau_i|_{i=0}^{n}) \triangleq \circledast_{i=0}^{n} \mathscr{C}_{OF}(E, \mathtt{n}_i : \tau_i) \vee \mathtt{Pi}(E, \mathtt{n}_i, \mathtt{undefined}, \_, \_)$

Reserved Fields
$\mathtt{ReservedFields}(E) = \circledast_{f \in \mathcal{R}}(E, f) \mapsto \mathtt{None}$

Object-Fields-Only
$\mathscr{C}_T(E, \{\ \mathtt{n}_i : \tau_i|_{i=0}^{n},\ \mathtt{n}'_j\ \mathtt{?} : \tau'_j|_{j=0}^{m}\ \}) \triangleq \mathtt{JSObjectGen}(E, \_, \_, \_) * \mathscr{C}_{OF}(E, \mathtt{n}_i : \tau_i|_{i=0}^{n})$
$* \mathscr{C}_{OF}(E, \mathtt{n}'_j\ \mathtt{?} : \tau'_j|_{j=0}^{m})$

Object-Fields-IndexSignature
$\mathscr{C}_T(E, \{\ \mathtt{n}_i : \tau_i|_{i=0}^{n},\ \mathtt{n}'_j\ \mathtt{?} : \tau'_j|_{j=0}^{m},\ [\mathtt{n} : \mathtt{string}] : \tau\ \}) \triangleq \mathtt{JSObjectGen}(E, \_, \_, \_) *$
$\mathscr{C}_{OF}(E, \mathtt{n}_i : \tau_i|_{i=0}^{n}) * \mathscr{C}_{OF}(E, \mathtt{n}'_j\ \mathtt{?} : \tau'_j|_{j=0}^{m}) *$
$\mathtt{IndexSig}_\tau(E, \#fields) * \mathtt{ReservedFields}(E)$

Object-CallSignature
$\mathscr{C}_T(E, \{\ (\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r,\ \mathtt{n}'_j : \tau'_j|_{j=0}^{m},\ \mathtt{n}''_k\ \mathtt{?} : \tau''_k|_{k=0}^{p}\ \}) \triangleq \mathtt{JSFunctionObject}(E, \_, \_)$
$* \mathscr{C}_{OF}(E, \mathtt{n}'_j : \tau'_j|_{j=0}^{m}) * \mathscr{C}_{OF}(E, \mathtt{n}''_k\ \mathtt{?} : \tau''_k|_{k=0}^{p})$

Object-ConstructorSignature
$\mathscr{C}_T(E, \{\ \mathtt{new}(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r,\ \mathtt{n}'_j : \tau'_j|_{j=0}^{m},\ \mathtt{n}''_k\ \mathtt{?} : \tau''_k|_{k=0}^{p}\ \}) \triangleq \mathtt{JSFunctionObject}(E, \_, \_)$
$* \mathscr{C}_{OF}(E, \mathtt{n}'_j : \tau'_j|_{j=0}^{m}) * \mathscr{C}_{OF}(E, \mathtt{n}''_k\ \mathtt{?} : \tau''_k|_{k=0}^{p})$

Index signature predicate
$\mathtt{IndexSig}_\tau(E_o, E_f) =$
$[\mathtt{base}]\, E_f = []$
$[\mathtt{recexists}]\, E_f = \#f :: \#other * \mathtt{DataProp}(E_o, \#f, \#v) * \mathscr{C}_T(\#v, \tau) * \mathtt{IndexSig}_\tau(E_o, \#other)$
$[\mathtt{recmissing}]\, E_f = \#f :: \#other * (E, \#f) \mapsto \mathtt{None} * \mathtt{IndexSig}_\tau(E_o, \#other)$

Note: All logical variables used in the rules above are considered to be fresh.

Primitive types. The rules Type-Void, Type-Undef, Type-Null, Type-Number, Type-String, Type-StringLiteral, and Type-Bool are straightforward. All these TypeScript types have a JavaScript equivalent which is preserved in JaVerT.

Any type. In separation logic, no resource equates no knowledge. The rule Type-Any states that we do not have any knowledge about a variable of type any.

Union type. The rule TYPE-UNION states that the logical expression E, must either be of type τ_1, or τ_2. The rule uses the $\vee$ operator, which is only part of the syntax of JaVerT assertions at the top level. To handle this aspect, after all assertions are generated, we write them in *disjunctive normal form* so they can be expressed in JaVerT syntax.

Object fields compiler function. In order to discuss the translation rules for objects, we define the auxiliary compiler $\mathscr{C}_{OF}$ through the rules MANDATORY FIELDS COMPILER and OPTIONAL FIELDS COMPILER. The assertion $\mathscr{C}_{OF}(E, \texttt{n}_i : \tau_i|_{i=0}^{k})$ states that all the fields in the set $\{\texttt{n}_0, \ldots, \texttt{n}_k\}$ are present in the object E and they have associated with them values satisfying types $\tau_0, \ldots, \tau_k$, respectively. To capture this for an object with an unknown prototype chain, we use the `Pi` predicate introduced in §2.2.3. The predicate $\texttt{Pi}(E, \texttt{n}, \#desc, \_, \_)$ says that accessing field `n` on the object E, resolves to the descriptor $\#desc$. The built-in predicate $\texttt{DescVal}(\#desc, \#v)$ extracts the value $\#v$ associated with the descriptor $\#desc$. Lastly, we require that the value $\#v$ has type τ_i, as indicated by the type declaration. The definition of $\mathscr{C}_{OF}$ for optional fields is very similar: it models as a disjunction the cases where a given field `n` is present or absent.

```
1  interface A {
2      a: string; b: string;
3  }
4  interface B { b: string; }
5  interface C {
6      a?: number; b: string;
7  }
8  var x: A = { a: "A", b: "B" };
9  var y: B = x;
10 var z: C = y;
11 z.a = 3;
```

FIGURE 3.1: TypeScript code that causes unsoundness.

Restricted fields. The prototype chain of every JavaScript object by default terminates with the `Object.prototype` object. We expect that the objects used in our program will not shadow any of the fields inherited from `Object.prototype`, as doing so would potentially break prototype safety. We define $\mathcal{R}$ to be the set of field names that are reserved for built-in objects and provide a predicate, `ReservedFields`, which takes as its single parameter an object location E and ensures that none of these reserved fields are defined in E. The predicate `ReservedFields` is defined in the rule RESERVED FIELDS.

Non-extensible objects. In the rule OBJECT-FIELDS-ONLY we express that:

1. At location E there is an object with an unknown prototype. We use the `JSObjectGen` predicate over the `JSObjWithProto` predicate, as we have no guarantees on the value of the internal property `@class`: E could be an object, or it could be a function—the type declaration states nothing about this.

2. The object at location E contains all the fields in the set $\{\texttt{n}_0, \ldots \texttt{n}_n\}$ and their types are $\tau_0, \ldots, \tau_n$ respectively. The fields in the set $\{\texttt{n}'_0, \ldots \texttt{n}'_m\}$ need not be present in the object; if a field $\texttt{n}'_i$ is present, it has type τ'_i. We use the auxiliary compiler $\mathscr{C}_{OF}$ to express these requirements.

3. We do not have any information about fields not in $\{\texttt{n}_0, \ldots, \texttt{n}_n, \texttt{n}'_0, \ldots, \texttt{n}'_m\}$. This is implicit in our translation. This causes our assignment compatibility relation to be different to that of TypeScript. We show in Figure 3.1 code that passes the TypeScript type checking process and leads to unsoundness. In TypeScript, we are able to assign an object with more fields to a variable whose type requires fewer fields. Hence, in our example, we are able to assign `x` to `y`. TypeScript keeps no information about the `y.a` field. Because from TypeScript's perspective field `a` does not exist in `y` and because field `a` is optional in `C`, we can assign `y` to `z`. Since `z` is of type `C`, if field `a` exists in `z`, it is expected to have type `number`. We are thus able to assign field `a` a number, via `z` (line 11. Since `x` and `z` refer to the same object,

after the assignment in line 11, `x.a` evaluates to a `number`, even though according to its type definition it should evaluate to a `string`. Using our translation, the assignment in line 10 fails verification, as we cannot prove that in the prototype chain of the object indicated by `y`, `a` is either absent or has type `number`.

Traditional JavaScript objects. In JavaScript it is common for objects to be dynamically accessed (i.e. `o[prop]` where `o` is an object and `prop` is a string denoting the name of the field that the programmer wants to retrieve). TypeScript only allows dynamic accesses when an index signature, `[key: string]:` τ, is present in the object type literal describing the object `o`. For objects with index signatures, any `string` `str` can be used for dynamic property access and if the field exists, it has associated a value of type τ. The predicate $\texttt{IndexSig}_\tau(E_o, E_f)$, says that the set of keys E_f may or may not be present in the object E_o; present keys are mapped to values of type τ. Modelling missing resource is common in separation logic. The set E_f denotes all possible keys. The definition of the predicate $\texttt{IndexSig}_\tau(E_o, E_f)$ is presented in the rule INDEX SIGNATURE PREDICATE.

Rule OBJECT-FIELDS-INDEXSIGNATURE translates object types which, besides named fields, have an index signature. As explained, the list of fields present in the object is existentially quantified via the $\#fields$ logical variable. We ensure that fields added via dynamic accesses do not break prototype safety via the `RestrictedFields` predicate.

When an object literal type has associated an index signature `[key: string]:` τ, the TypeScript standard requires that the type τ is the same as the type of all named fields. The OBJECT-FIELDS-INDEXSIGNATURE rule allows for more flexibility. This is possible because during verification, the code is executed symbolically and we can determine whether dynamic field accesses are guaranteed not to target any of the explicitly declared fields. While this would be a useful feature, implementing it would mean that some programs which would fail `tsc` compilation would be successfully verified using our software. This is an issue, because we use `tsc` extensively in our compilation. To support checking programs that fail `tsc` compilation, we would have to implement the checks `tsc` is performing in our software, thereby increasing its complexity.

Objects with call or constructor signatures. The rules OBJECT-CALLSIGNATURE and OBJECT-CONSTRUCTORSIGNATURE are identical as the memory footprint of regular functions and constructors is identical. The difference between the two is semantic: objects with constructor signatures must be called using the `new` keyword, while objects with call signatures must *not* be called using the `new` keyword. In translating these two types to assertions, we make use of the `JSFunctionObject` JaVerT built-in predicate. Since functions are simply special objects in JavaScript, they may have named fields associated with them; the translation of named fields is the same as for normal objects.

3.1.1 Translating the typing environment without classes

Given the the non-overlapping resources assumption and the fact that we are currently working with a typing environment where no `class` types are present, we extend our translation to the level of typing environments. The compiler $\mathscr{C}_{TE}$ translates a typing environment Γ to a JaVerT assertion specifying the type of all variables. It is defined by the rule TS-ENV given below.

Translation of TypeScript type environment to JaVerT assertion, without `class` types

TS-ENV

$\mathscr{C}_{TE}(\Gamma) \triangleq \circledast_{\texttt{x}:\,\tau \in \Gamma} \texttt{Scope}(\texttt{x}, \#x) * \mathscr{C}_T(\#x, \tau)$

Note: The logical variable used is considered to be fresh in every conjunct.

3.1.2 Specifying functions without classes

JaVerT requires users to provide pre- and post-conditions for the functions they define. For every function `f` we generate the specification based on the rules in the table below.

Auxiliary functions. We define $\Gamma_{in}(\mathtt{f})$ to be the typing environment containing the parameters of the function `f`, and $\Gamma_{out}(\mathtt{f})$ to be the typing environment of variables used in the function `f`, but declared in an outer scope (i.e. captured variables). These two sets of variables associated with types can be determined statically—we use the *captured* and *params* functions to denote the process of determining the two sets. The function *return* maps every function to its declared return type.

Function pre- and post-condition compile rules: $\mathscr{C}^F_{pre}$ **and** $\mathscr{C}^F_{post}$

FUNC-PRE-NOCLASS

$$\frac{\Gamma_{out}(\mathtt{f}) = captured(\mathtt{f}) \quad \Gamma_{in}(\mathtt{f}) = params(\mathtt{f})}{\mathscr{C}^F_{pre}(\mathtt{f}) \triangleq \mathscr{C}_{TE}(\Gamma_{out}(\mathtt{f})) * \circledast_{(\mathtt{x}:\tau)\in\Gamma_{in}(\mathtt{f})}\mathscr{C}_T(\mathtt{x},\tau)}$$

FUNC-POST-NOCLASS

$$\frac{A_{pre} = \mathscr{C}^F_{pre}(\mathtt{f}) \qquad \tau_r = return(\mathtt{f})}{\mathscr{C}^F_{post}(\mathtt{f}) \triangleq A_{pre} * \mathscr{C}_T(\mathtt{ret},\tau_r)}$$

FUNC-SPEC-NOCLASS

$$\frac{A_{pre} = \mathscr{C}^F_{pre}(\mathtt{f}) \quad A_{post} = \mathscr{C}^F_{post}(\mathtt{f})}{\mathscr{C}^F_{spec}(\mathtt{f}) \triangleq \left\{\{A_{pre}\}\mathtt{f}\{A_{post}\}\right\}}$$

Pre-condition rule. Since $\Gamma_{out}(\mathtt{f})$ is a subset of the outer typing environment, Γ, we can use the $\mathscr{C}_{TE}$ function to obtain the assertion corresponding to the captured variables. To ensure the parameters passed in have the correct type, applications of $\mathscr{C}_T$ suffice. Using the $\mathscr{C}_{TE}$ for parameters, as for captured variables, would needlessly impose the use of the JaVerT built-in `Scope` for getting the value of the parameters, which we know exist in the current scope. These judgements lead to the FUNC-PRE-NOCLASS rule.

Post-condition rule. In the FUNC-POST-NOCLASS rule, we make use of the pre-condition of the function and generate an extra conjunct for the `ret`, imposing that it has the required type τ_r. We are able to derive the post-condition in this manner due to the invariant nature of types in TypeScript—once a variable `x` is declared to have type τ, it cannot change its type. Hence, all captured variables and all parameters must have the same type after the function's body is executed, as they did before.

The spec rule. The FUNC-SPEC-NOCLASS assembles the generated pre-/post-conditions, generating a set of specifications for the `f` function. In the case of the present rule, the set always contains one element.

Specifying methods and constructors. Methods and constructors need to be specified as well. For those, reasoning about the type of the `this` object is required and hence they are covered after we introduce the translation for classes.

3.2 The translation with classes

This section extends the translation to the level of classes. Classes provide us with complete knowledge of their prototype chain statically. They also come coupled with the challenge of carrying out the translation from TypeScript to ES5, since ES5 does not support classes. As mentioned in §2.1, classes represent syntactic sugar for prototype-based inheritance and do not introduce an object-oriented paradigm to JavaScript.

3.2.1 Relevant design decisions for translating classes

Before describing the translation for classes, we aim to get an intuitive grasp of what the structure of an object of type `C` is, where `C` is an arbitrary class. We take into consideration two main aspects: (1) the representation of the prototype chain structure; (2) which fields belong in instances and which fields belong in prototypes.

```
1  class C {
2    f(): void {
3      console.log("BCD");
4    }
5  }
6  function g(x: C): void {
7    x.f();
8  }
9  g({
10   f: function(): void {
11     console.log("xcd");
12   }
13 });
```

FIGURE 3.2: TypeScript example illustrating the assignment relation for classes.

Enforcing the prototype chain structure. To address the first point, we introduce the following scenario via the example in Figure 3.2: a function `g` takes an argument `x`, of type `C`, which should provide one method, `f`. Most developers would expect the definition of `f` to be the one in the class `C`. The TypeScript compiler, `tsc`, type checks if an object said to have type `C` can resolve all the fields a programmer would expect to find in an instance of class `C`, either as its own fields, or through its prototype chain. Hence, `tsc` allows the object literal in between lines 9-13 to be assigned to a variable of type `C` despite the fact that the prototype chain does not contain `C.prototype`. This different structure of the prototype chains means that the definition of `f` is not the one in class `C`. We depart from TypeScript in that we require an object of type `C` to preserve the structure of the prototype chain, as defined at class declaration. We find this extra requirement to be in accordance with the programmers' intuition.

Prototype safety revisited. We describe each class `C` using two predicates: one describing its prototype, and one describing each instance. When stating these, our goal is to ensure prototype safety, defined in [17]: "The specification of a given library must ensure that all prototype chains are consistent with correct library behaviour by stating which resources must not be present for its code to run correctly. In particular, (P1) constructed objects cannot redefine properties that are to be found in their prototypes; and (P2) prototypes cannot define as non-writable those properties that are to be present in their instances. We refer to these two criteria as prototype safety". Hence, when expressing the predicate corresponding to the prototype, we need to ensure that the prototype does not define as non-writable properties that are to be present in their instances. A stronger assertion is requiring that the prototype does not define properties that are to be present in their instances *at all*. Similarly, when defining the predicate corresponding to instances, we want to ensure that objects do not redefine properties that are to be found in their prototypes.

Distributing fields between instance and prototype. We follow the choices of `tsc` to determine which properties belong in instances and which ones belong prototypes. The TypeScript compiler places methods, defined as $\underline{\kappa}\ \mathtt{x}(\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \{\ \overline{\mathtt{s}}\ \}$, in class prototypes and all other fields in class instances. We introduce the following notation to refer to the fields in an object and to the methods defined by a class:

1. We denote the set of all field names present in an instance of class `A` by $\mathcal{F}_{\mathtt{A}}$. In Figure 3.3, we have $\mathcal{F}_{\mathtt{A}} = \mathcal{F}_{\mathtt{B}} = \mathcal{F}_{\mathtt{C}} = \{\texttt{"x"}\}$; an instance of a class inherits all the fields defined in both the class and its superclasses;

```
class A {
  x: number = 3;
  a(): void {
    console.log("a");
  }
}
class B extends A {
  b(): void {
    console.log("b");
  }
}
class C extends A {
  c(): void {
    console.log("c");
  }
}
```

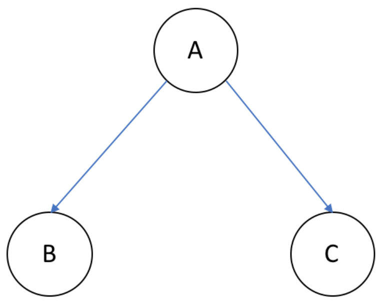

FIGURE 3.3: Example of class inheritance in TypeScript (left) and the associated inheritance graph (right)

2. We denote the set of all methods defined in class `A` by $\mathcal{N}_{\mathtt{A}}$. In Figure 3.3, we have $\mathcal{N}_{\mathtt{A}} = \{\texttt{"a"}\}$, $\mathcal{N}_{\mathtt{B}} = \{\texttt{"b"}\}$ and $\mathcal{N}_{\mathtt{C}} = \{\texttt{"c"}\}$. For any class `C` all members of $\mathcal{N}_{\mathtt{C}}$ are expected to be found in `C.prototype`.

3.2.2 The inheritance graph and its relation to prototype safety

We model an inheritance graph as follows: classes are represented as nodes; introduce a directed edge from node `A` to node `B` if class `B` extends class `A`. In a valid TypeScript program there are no circular inheritance chains; `tsc` performs this check. Hence, the inheritance graph is a directed acyclic graph (DAG). In the table below we introduce functions for describing classes from the perspective of the inheritance graph.

Definition of prototype-safety relevant sets

CLASS-ALLDESCENDANTS

$$\frac{\mathcal{C} = \mathit{descendants}(\mathtt{C})}{\mathit{descendants}^{+}(\mathtt{C}) \triangleq \{\mathtt{C}\} \cup \bigcup_{\mathtt{C}\in\mathcal{C}} \mathit{descendants}^{+}(\mathtt{C})}$$

CLASS-ALLPARENTS-BASE

$$\frac{\mathit{ancestor}(\mathtt{C}) = \bot}{\mathit{ancestor}^{+}(\mathtt{C}) \triangleq \{\mathtt{C}\}}$$

CLASS-ALLPARENT-INHERIT

$$\frac{\mathit{ancestor}(\mathtt{C}) = \mathtt{P} \quad \mathcal{P} = \mathit{ancestor}^{+}(\mathtt{P})}{\mathit{ancestor}^{+}(\mathtt{C}) \triangleq \{\mathtt{C}\} \cup \mathcal{P}}$$

CLASS-ALLMETHODS

$$\frac{\mathcal{C} = \mathit{ancestor}^{+}(\mathtt{C})}{\mathcal{N}^{+}_{\mathtt{C}} \triangleq \bigcup_{C\in\mathcal{C}} \mathcal{N}_{C}}$$

CLASS-ALLFIELDS

$$\frac{\mathcal{C} = \mathit{descendants}^{+}(\mathtt{C})}{\mathcal{F}^{+}_{\mathtt{C}} \triangleq \bigcup_{C\in\mathcal{C}} \mathcal{F}_{C}}$$

Descendants and parents. We introduce the functions *descendants* and *ancestor*, which take a class `C` as argument and return its *direct* descendants or ancestor, respectively. In the example given in Figure 3.3 we have that $\mathit{descendants}(\mathtt{A}) = \{\mathtt{B},\mathtt{C}\}$, $\mathit{descendants}(\mathtt{B}) = \mathit{descendants}(\mathtt{C}) = \varnothing$, $\mathit{ancestor}(\mathtt{A}) = \bot$, and $\mathit{ancestor}(\mathtt{B}) = \mathit{ancestor}(\mathtt{C}) = \mathtt{A}$. The rule CLASS-ALLDESCENDANTS defines $\mathit{descendants}^{+}$, a function which gives us all the descendants of a given class `C`, not necessarily direct. Rules CLASS-ALLPARENTS-BASE and CLASS-ALLPARENTS-INHERIT define $\mathit{ancestor}^{+}$, a function which provides us with all of the ancestors of a class `C`. For convenience, we include the class `C` in the sets $\mathit{descendants}^{+}(\mathtt{C})$ and $\mathit{ancestor}^{+}(\mathtt{C})$. For the inheritance graph in Figure 3.3, we have the following: $\mathit{descendants}^{+}(\mathtt{A}) = \{\mathtt{A},\mathtt{B},\mathtt{C}\}$, $\mathit{descendants}^{+}(\mathtt{B}) = \{\mathtt{B}\}$, $\mathit{descendants}^{+}(\mathtt{C}) = \{\mathtt{C}\}$, $\mathit{ancestor}^{+}(\mathtt{A}) = \{\mathtt{A}\}$, $\mathit{ancestor}^{+}(\mathtt{B}) = \{\mathtt{A},\mathtt{B}\}$, $\mathit{ancestor}^{+}(\mathtt{C}) = \{\mathtt{A},\mathtt{C}\}$.

First prototype safety condition. To ensure prototype safety, it is required that instances do not redefine properties that are to be found in their prototypes. We previously established that named functions (i.e. methods) are meant to be found in the prototype chain, not in instances. The set of methods found in the prototype chain for an instance of a given class `C` is given by $\mathcal{N}_{\texttt{C}}^{+}$. In our assertions, we ensure that only elements of $\mathcal{F}_{\texttt{C}}$ are present in instances of `C` via the $\mathsf{emptyFields}(E \mid \mathcal{F}_{\texttt{C}})$, which states that the object at the location denoted by E has no properties other than possibly those included in the set denoted by $\mathcal{F}_{\texttt{C}}$. Since `tsc` checks that $\mathcal{F}_{\texttt{C}} \cap \mathcal{N}_{\texttt{C}}^{+} = \varnothing$, our requirement guarantees prototype safety. The set $\mathcal{N}_{\texttt{C}}^{+}$ can be determined statically using the rule Class-AllMethods. In the example in Figure 3.3, we have $\mathcal{N}_{\texttt{A}}^{+} = \{\texttt{"a"}\}$, $\mathcal{N}_{\texttt{B}}^{+} = \{\texttt{"a"}, \texttt{"b"}\}$, $\mathcal{N}_{\texttt{C}}^{+} = \{\texttt{"a"}, \texttt{"c"}\}$.

Second prototype safety condition. Prototype safety requires that prototypes do not define as non-writable those properties that are to be present in their instances. As per our discussion, we strengthen this requirement such that fields that are to be in the instances, are not defined at all in the prototype chain. There are two aspects we are concerned with:

1. **Fields of class instances.** In the rule Class-AllFields, we define $\mathcal{F}_{\texttt{C}}^{+}$ as the set of fields present in instances of `C` or any of its subclasses. The prototypes corresponding to any class `C` must not declare as non-writable any of the fields $f \in \mathcal{F}_{\texttt{C}}^{+}$. For the code in Figure 3.3, we have $\mathcal{F}_{\texttt{A}}^{+} = \mathcal{F}_{\texttt{B}}^{+} = \mathcal{F}_{\texttt{C}}^{+} = \{\texttt{"x"}\}$. If `A.prototype` defined as non-writable property `x`, we would be unable to create instances of `A`, `B`, and `C`.

2. **Methods of subclasses.** In the example presented in Figure 3.3, if `A.prototype` would have declared property `b` as non-writable, then the declaration of `class B` would have been illegal. Since prototypes are just regular objects and `A.prototype` is the parent of `B.prototype` along the prototype chain, this declaration of `b` as non-writable would break prototype safety.

To ensure that neither of the two cases occur, we require that the prototypes of the classes defined only contain the properties corresponding to the class methods. Furthermore, these properties must be writable in order to ensure methods can be overriden in subclasses.

3.2.3 The `class` JaVerT predicates

For every class `C`, we define four predicates:

1. $\mathit{Proto}_{\texttt{C}}(p, \mathit{pproto}, \mathit{sch})$, describing what it means for p to have the shape expected for C`.prototype`; pproto represents the next object along the prototype chain, and sch is the scope chain in which the methods are defined.

2. $\mathit{Instance}_{\texttt{C}}(x, p)$, describing each instance x of the class C; here, p represents the C`.prototype` object.

3. $\mathit{Constructor}_{\texttt{C}}(\mathit{constr}, p, \mathit{sch})$, describing the constructor constr of the class C; p represents the `prototype` field of the constructor (i.e. not the internal `proto` property), and sch is the scope chain in which the constructor was declared.

4. $\mathit{ProtoAndConstructor}_{\texttt{C}}(p, \mathit{pproto}, \mathit{sch})\mathit{constr}$, bundling the resources corresponding to the prototype p and the constructor constr. The pproto parameter represents the next object along the prototype chain, while sch is the scope chain in which the methods of class C and its constructor were defined.

These predicates are defined below.

JaVerT predicates for class prototypes and instances

Class-ProtoLVar

$$\frac{c = name(C)}{pvn(C) \triangleq \texttt{"\#"} \texttt{ ++ } c \texttt{ ++ } \texttt{"proto"}}$$

Class-ConstrLVar

$$\frac{c = name(C)}{cvn(C) \triangleq \texttt{"\#"} \texttt{ ++ } c}$$

Class-ScopeLVar

$$\frac{c = name(C)}{svn(C) \triangleq \texttt{"\#"} \texttt{ ++ } c \texttt{ ++ } \texttt{"sch"}}$$

Class-Prototype

$$\begin{aligned} Proto_C(p, pproto, sch) \triangleq\ & \texttt{JSObjWithProto}(p, pproto) * \\ & \circledast_{m \in \mathcal{N}_C} \left[\texttt{DataProp}(p, m, \#m\_loc) * \texttt{JSFunctionObjectStrong}(\#m\_loc, Cm, sch, \_)\right] * \\ & \mathsf{emptyFields}(p \mid \mathcal{N}_C) \end{aligned}$$

Class-Instance

$$\begin{aligned} Instance_C(x, p) \triangleq\ & \texttt{JSObjWithProto}(x, p) * \circledast_{f \in \mathcal{F}_C} \left[\texttt{DataProp}(x, f, \#v) * \mathscr{C}_T(\#v, type_C(f))\right] * \\ & \mathsf{emptyFields}(x \mid \mathcal{F}_C) \end{aligned}$$

Class-Constructor

$$Constructor_C(constr, p, sch) \triangleq \texttt{JSFunctionObjectStrong}(constr, C\_constr, sch, p)$$

Class-ProtoAndConstructor-NoInheritance

$$\frac{ancestor(C) = \bot}{\begin{aligned} ProtoAndConstructor_C(p, sch, constr) \triangleq\ & \texttt{Scope}(C\_constr, C : constr, sch) * \\ & Proto_C(p, \texttt{Object.prototype}, sch) * Constructor_C(constr, p, sch) \end{aligned}}$$

Class-ProtoAndConstructor-WithInheritance

$$\frac{ancestor(C) = A \qquad pproto = pvn(A) \qquad pconstr = cvn(A)}{\begin{aligned} ProtoAndConstructor_C(p, sch, constr) \triangleq\ & \texttt{Scope}(C\_constr, C : constr, sch) * \\ & Proto_C(p, pproto, sch) * Constructor_C(constr, p, sch) * \\ & \texttt{Scope}(C\_constr, \texttt{\_super} : pconstr, sch) \end{aligned}}$$

Prototype, constructor and scope chain logical variable names. We associate with each class a set of three logical variables corresponding to their prototype, constructor and scope chain. The names of these logical variables is determistic and obtained using the auxiliary functions *pvn*, *cvn*, and *svn*, defined via the rules Class-ProtoLVar, Class-ConstrLVar, and Class-ScopeLVar, respectively.

The prototype predicate. The rule Class-Prototype models the inheritance relation by allowing the parameter *pproto*, the parent on the prototype chain be passed in as a parameter. If the class C does not explicitly `extend` any other class, we consider *pproto* to be `Object.prototype`. In TypeScript, class methods cannot have user-defined properties, unlike regular function objects. We use the `JSFunctionObjectStrong` JaVerT built-in predicate (§2.2.3) to capture this. All methods $m \in \mathcal{N}_C$ are defined in the same scope chain; hence, we use the same *sch* variable for all methods. We highlight in pink the prototype safety requirement for prototypes, (P2).

The instance predicate. The $Instance_C(x, p)$ predicate, which is defined in the rule Class-Instance, takes the instance and the prototype objects as parameters. We use `DataProp` for describing the fields instead of $\mathscr{C}_{OF}$. Using $\mathscr{C}_{OF}$ would impose a loose specification in the assertion it outputs: a field can be found anywhere in the prototype chain. In the context of classes, we know the fields in the set $\mathcal{F}_C$ are to be found in the instance itself and using the `Pi` predicate as part of $\mathscr{C}_{OF}$ would not capture that. The prototype safety requirement for instances, (P1), is highlighted in pink.

The constructor predicate. We define the $Constructor_C(constr, p, sch)$ predicate in rule CLASS-CONSTRUCTOR; it takes the constructor, its prototype field, and the scope chain in which it is defined as parameters.

The prototype and constructor predicate. We define the *ProtoAndConstructor* predicate, which bundles together the resources associated with the prototype and the constructor of class C, in the rules CLASS-PROTOANDCONSTRUCTOR-NOINHERITANCE and CLASS-PROTOANDCONSTRUCTOR-WITHINHERITANCE. These two definitions deal with the case when the class C is a base class or a derived class, respectively. The predicate takes as arguments:

1. p, the C`.prototype` object;
2. sch, the scope chain in which the constructor/methods of class C were defined;
3. $constr$, the constructor object.

We take advantage of the deterministic mapping from classes to logical variable names and omit the parent along the prototype chain and the constructor of the parent class from the list of parameters.

It is essential to understand that for any class C, its methods and the constructor are defined in the same scope chain; this can be observed by analysing the translation from classes to ES5 code in §2.1.5. We illustrate the importance of this by an example in §3.3. The translation of derived classes uses a `_super` parameter (§2.1.5), bound to the constructor of the parent class. The *ProtoAndConstructor* predicate captures this binding of the `_super` parameter in the scope chain sch via the conjunct highlighted in pink.

3.2.4 The definitions of the compile functions revisited

We revisit the definitions for all five compilers in the scenario where `class` types are present. The updated definitions are presented in the table below:

The definition of all compile functions with classes

TYPE-CLASS

$$\frac{\mathcal{C} = descendants^{+}(\texttt{C})}{\mathscr{C}_T(E, \texttt{C}) \triangleq \bigvee_{C \in \mathcal{C}} Instance_C(E, pvn(C))}$$

PROGRAM-PROTOSANDCONSTRUCTORS

$$\frac{}{\begin{array}{l}\texttt{AllProtos}(\mathcal{C}) \triangleq \texttt{ObjectPrototype()} * \texttt{FunctionPrototype()} * \texttt{GlobalObject()} * \\ \quad \circledast_{C \in \mathcal{C}} ProtoAndConstructor_C(pvn(C), svn(C), cvn(C)) * \texttt{GlobalVar}(C, cvn(C))\end{array}}$$

FUNC-PRE

$$\frac{\Gamma_{out}(\texttt{f}) = captured(\texttt{f}) \quad \Gamma_{in}(\texttt{f}) = params(\texttt{f})}{\mathscr{C}^F_{pre}(\texttt{f}) \triangleq \texttt{AllProtos}(\mathcal{C}) * \mathscr{C}_{TE}(\Gamma_{out}(\texttt{f})) * \circledast_{(\texttt{x}:\tau) \in \Gamma_{in}(\texttt{f})} \mathscr{C}_T(\texttt{x}, \tau)}$$

FUNC-POST

$$\frac{A_{pre} = \mathscr{C}^F_{pre}(\texttt{f}) \qquad \tau_r = return(\texttt{f})}{\mathscr{C}^F_{post}(\texttt{f}) \triangleq A_{pre} * \mathscr{C}_T(\texttt{ret}, \tau_r)}$$

FUNC-SPEC

$$\frac{A_{pre} = \mathscr{C}^F_{pre}(\texttt{f}) \quad A_{post} = \mathscr{C}^F_{post}(\texttt{f})}{\mathscr{C}^F_{spec}(\texttt{f}) \triangleq \left\{ \{A_{pre}\}\texttt{f}\{A_{post}\} \right\}}$$

Method-Pre

$$\frac{\Gamma_{out}(\texttt{m}_\texttt{C}) = \mathit{captured}(\texttt{m}_\texttt{C}) \qquad \Gamma_{in}(\texttt{m}_\texttt{C}) = \mathit{params}(\texttt{m}_\texttt{C}) \qquad D \in \mathit{descendants}^+(\texttt{C})}{\begin{array}{l}\mathscr{C}^M_{pre}(\texttt{m}_\texttt{C}, D) \triangleq \texttt{AllProtos}(\mathcal{C} \setminus \{\texttt{C}\}) * \mathit{Instance}_D(\texttt{this}, \mathit{pvn}(D)) * \\ \quad \mathit{ProtoAndConstructor}_C(\mathit{pvn}(C), \texttt{\$\$scope}, \mathit{cvn}(C)) * \\ \quad \mathscr{C}_{TE}(\Gamma_{out}(\texttt{m}_\texttt{C})) * \circledast_{(\texttt{x}:\tau) \in \Gamma_{in}(\texttt{m}_\texttt{C})} \mathscr{C}_T(\texttt{x}, \tau)\end{array}}$$

Method-Post

$$\frac{D \in \mathit{descendants}^+(\texttt{C}) \qquad A_{pre} = \mathscr{C}^M_{pre}(\texttt{m}_\texttt{C}, D) \qquad \tau_r = \mathit{return}(\texttt{m}_\texttt{C})}{\mathscr{C}^M_{pre}(\texttt{m}_\texttt{C}, D) \triangleq A_{pre} * \mathscr{C}_T(\texttt{ret}, \tau_r)}$$

Method-Spec

$$\frac{}{\mathscr{C}^M_{spec}(\texttt{m}_\texttt{C}) \triangleq \left\{\{\mathscr{C}^M_{pre}(\texttt{m}_\texttt{C}, D)\}\texttt{m}_\texttt{C}\{\mathscr{C}^M_{post}(\texttt{m}_\texttt{C}, D)\} | D \in \mathit{descendants}^+(\texttt{C})\right\}}$$

Constructor-Pre

$$\frac{\begin{array}{c}\Gamma_{out}(\texttt{constr}_\texttt{C}) = \mathit{captured}(\texttt{constr}_\texttt{C}) \quad \Gamma_{in}(\texttt{constr}_\texttt{C}) = \mathit{params}(\texttt{constr}_\texttt{C}) \\ \mathit{proto} \in \{\mathit{pvn}(C) | C \in \mathit{descendants}^+(\texttt{C})\}\end{array}}{\begin{array}{l}\mathscr{C}^C_{pre}(\texttt{C}, \mathit{proto}) \triangleq \texttt{AllProtos}(\mathcal{C} \setminus \{\texttt{C}\}) * \mathit{ProtoAndConstructor}_C(\mathit{pvn}(C), \texttt{\$\$scope}, \mathit{cvn}(C)) * \\ \quad \texttt{JSObjWithProto}(\texttt{this}, \mathit{proto}) * \mathsf{emptyFields}(\texttt{this} \mid \varnothing) * \\ \quad \mathscr{C}_{TE}(\Gamma_{out}(\texttt{constr}_\texttt{C})) * \circledast_{(\texttt{x}:\tau) \in \Gamma_{in}(\texttt{constr}_\texttt{C})} \mathscr{C}_T(\texttt{x}, \tau)\end{array}}$$

Constructor-Post

$$\frac{\begin{array}{c}\Gamma_{out}(\texttt{constr}_\texttt{C}) = \mathit{captured}(\texttt{constr}_\texttt{C}) \quad \Gamma_{in}(\texttt{constr}_\texttt{C}) = \mathit{params}(\texttt{constr}_\texttt{C}) \\ \mathit{proto} \in \{\mathit{pvn}(C) | C \in \mathit{descendants}^+(\texttt{C})\}\end{array}}{\begin{array}{l}\mathscr{C}^C_{post}(\texttt{C}, \mathit{proto}) \triangleq \texttt{AllProtos}(\mathcal{C} \setminus \{\texttt{C}\}) * \mathit{ProtoAndConstructor}_C(\mathit{pvn}(C), \texttt{\$\$scope}, \mathit{cvn}(C)) * \\ \quad \mathit{Instance}_\texttt{C}(\texttt{this}, \mathit{proto}) * \mathscr{C}_{TE}(\Gamma_{out}(\texttt{constr}_\texttt{C})) * \circledast_{(\texttt{x}:\tau) \in \Gamma_{in}(\texttt{constr}_\texttt{C})} \mathscr{C}_T(\texttt{x}, \tau)\end{array}}$$

Constructor-Spec

$$\frac{\mathcal{D} = \{\mathit{pvn}(C) | C \in \mathit{descendants}^+(\texttt{C})\}}{\mathscr{C}^C_{spec}(\texttt{constr}_\texttt{C}) \triangleq \left\{\{\mathscr{C}^C_{pre}(\texttt{constr}_\texttt{C}, \mathit{proto})\}\texttt{constr}_\texttt{C}\{\mathscr{C}^C_{post}(\texttt{constr}_\texttt{C}, \mathit{proto})\} | \mathit{proto} \in \mathcal{D}\right\}}$$

Translating class types. An instance of a class `C` is translated following the rule Type-Class. Subclassing is supported by iterating over all the classes that have `C` as an ancestor. This translation of class types breaks modularity: for every new class that extends class `C`, we need to reverify all specifications which mention `C`. This is a consequence of the lack of higher-order reasoning in JaVerT; we discuss this in §3.2.5.

All prototypes and constructors. We define the `AllProtos`$(\mathcal{C})$ predicate to specify the resources corresponding to the prototypes and constructors of all classes in the set $\mathcal{C}$. In this context $\mathcal{C}$ is taken as argument and does not refer to the set of all classes in the program, as defined in §2.1.1. The predicate specifies the resources associated with `Object.prototype`, `Function.prototype` and the global object; it is reasonable to expect these resources to be available as JavaScript code frequently relies on function defined in these two prototypes and on accessing fields in the global object. The prototypes and constructors associated with all the classes $C \in \mathcal{C}$ are specified via a conjunction of *ProtoAndConstructor*. As discussed earlier, *ProtoAndConstructor* requires knowledge of the constructor resource: we resolve the constructors' resources via the global object using the `GlobalVar` predicate.

Function specification. The rule Func-Pre only differs from Func-Pre-NoClass in that it adds the `AllProtos`$(\mathcal{C})$ conjunct, highlighted in pink, providing the resources associated with the constructors and prototypes of all classes in the program. These resources are needed to operate with existent class instances or create new ones. The post-condition can be obtained from the pre-condition using the rule FUNC-POST. This rule is identical to FUNC-POST-NOCLASS. If τ_r is a class type, we are guaranteed to have available all the resources needed because in the pre-condition we specify the structure of all prototypes and constructors. The pre- and post-conditions are put together using the FUNC-SPEC rule.

Method specification. The METHOD-SPEC rule relies on $\mathscr{C}^M_{pre}$ and $\mathscr{C}^M_{post}$ compilers defined in the rules METHOD-PRE and METHOD-POST, respectively. These are similar to FUNC-PRE and FUNC-POST used for regular functions. The difference is highlighted in pink. There are three conjuncts which are different to the ones in FUNC-PRE:

1. We use the `AllProtos` predicate only for the set $\mathcal{C} \setminus \{\texttt{C}\}$. We generate the predicate bundling the prototype and constructor of the `C` function separately.
2. The first extra conjunct requires that `this` is an instance of `C` or an instance of a descendant class of `C`. This is standard behaviour for subclassing: if a method is not defined in a class, we inspect the parent class and progress through the prototype chain until the method is found. The class which `this` is an instance of needs to be passed as a parameter to enable us to form groups of pre-/post-conditions where `this` does not change its type. This decision is imposed by the fact that assignments to `this` are forbidden in JavaScript.
3. The prototype and constructor of the `C` class are bundled together by an extra conjunct. The difference between the current usage of *ProtoAndConstructor* and its usage in the `AllProtos` predicate is that here we indicate that the scope in which the method is defined is the current scope, denoted by the `$$scope` JaVerT variable.

Iteration over all descendants of the current class, `C`, is done in the METHOD-SPEC rule.

Constructor specification. The rules CONSTRUCTOR-PRE and CONSTRUCTOR-POST define the pre- and the post-conditions for the constructor of a given class `C`. We highlight in pink the ways in which these rules differ from those describing the specifications of methods. In the pre-condition of the constructor, we require `this` to be a fresh object which has the prototype *proto*. The prototype of `this`, *proto*, is conditioned to be a prototype of one of the descendants of the class `C`. We take the prototype as a parameter instead of using the `C.prototype` object because constructors can be called from within a subclass' constructor; for that case, the prototype is that of the subclass, not `C.prototype`. We exemplify this in §3.3. In the post-condition we require `this` to have the expected shape of an instance of `C`, but we use the same *proto* variable as in the pre-condition, passed an argument. The rule CONSTRUCTOR-SPEC assembles the pre- and post-conditions, iterating through all the possible values of *proto*, in a similar fashion to the METHOD-SPEC rule.

3.2.5 Discussion

When a variable `x` is declared to have type `C`, it can refer to an instance of any class that has `C` as an ancestor. We model this through a disjunction over all the members of the $\textit{descendants}^+(\texttt{C})$ set in rule TYPE-CLASS. This leads to two main issues:

1. We face a state explosion problem: the number of specifications that JaVerT needs to verify grows exponentially with the number of disjunctions. Given the current translation of classes, we introduce a disjunction for every variable whose declared type is a class with descendants;

2. We lose modularity: if a new class is declared and it extends `C`, either directly or indirectly, we need to verify all the specifications in which variables of type `C` appear, including the methods and the constructor of `C`. This means that we cannot use our toolchain to verify a library and then distribute it. A client extending a class provided by the library would need to reverify the whole library.

```
1   class A {
2     constructor() { }
3     m(): string {
4       return "Some string";
5     }
6   }
7   class B extends A {
8     constructor() { super(); }
9     m(): string {
10      return "Some other string";
11    }
12  }
13  function f(x: A): string {
14    return x.m();
15  }
```

FIGURE 3.4: TypeScript method overriding

The root cause of the inefficient translation is the lack of higher-order reasoning in JaVerT: in order to call a function, we must know its precise identifier. Since classes can redefine methods defined in super-classes, we cannot precisely determine the identifier of a given method `m`, unless we have complete information on the structure of the prototype chain. In Figure 3.4, the method `m` defined in class `A`, is a property in `A.prototype` and has id `A_m`, while the overriding method `m`, defined in class `B`, is a property in `B.prototype` and has id `B_m`. The method `m` of `x` that is called in line 14, can resolve to either of these two methods, depending on whether `x` is an instance of class `A` or an instance of class `B`.

If JaVerT supported higher-order reasoning, we could potentially redefine the predicate describing the prototype of `C` such that it would not require methods to be present in one precise prototype object, but instead allow them to be resolved by inspecting the prototype chain. In this way, we would be able to provide a more efficient translation for classes. However, this approach is speculative as it is not clear how higher-order reasoning would work: since we do not know which function we are calling, we also do not know what resource it needs. Whereas it is easy to frame off [39] excess resource, it is much more difficult to anticipate the resource we might need in a higher-order setting. The reasoning would also become more complicated, especially given the fact that JavaScript can capture variables via closures.

3.3 Assertion placement — an example

To enable verification, we must place the JaVerT assertions generated by translating TypeScript type annotations in the emitted JavaScript program. We identify four different types of assertions that need to be laid out throughout our program:

1. **Predicate definitions**, such as $Proto_C(p, pproto, sch)$, must be placed at the top of the program; they do not interact with the emitted code in anyway. We generate predicates for every class declaration, instance declaration, index signature.

2. **Function, method and constructor specifications** describe the pre-conditions and post-conditions corresponding to every function, method and constructor in the program and are generated in accordance with the FUNC-SPEC, METHOD-SPEC and CONSTRUCTOR-SPEC rules;

3. **Assignment assertions** ensure that after every assignment the type of the value assigned to is preserved. Their placement depends on the interpretation of types we opt for; a discussion of the possibilities is presented shortly;

4. **Loop invariants** ensure that all the variables within the typing environment maintain their type before the loop and after every iteration of it;

We illustrate the way we place these assertions by analysing the JaVerT translation corresponding to the TypeScript code presented in Figure 2.6: the JaVerT translation of the `Animal` class is presented in Figure 3.5, while the `Cat` class is translated in Figure 3.6.

3.3.1 Predicate definitions

We first define the following predicates: `Animal`, `AnimalProto`, `AnimalConstructor`, and `AnimalProtoAndConstructor`, together with their `Cat` equivalents. In addition, the `AllProtosAndConstructors` predicate bundles together the prototypes and constructors associated with all the classes in the program.

3.3.2 Method and constructor specifications

Method specifications. We turn our attention to the only method in the `Animal` class, `walk` (lines 24-47 in Figure 3.5). Its two sets of pre-/post-conditions correspond to the cases when the `this` object is an instance of the `Animal` class or the `Cat` class respectively. The `Animal` and `Cat` classes are the only elements of the *descendants*$^+$(`Animal`) set; if a new class extended `Animal` we would need to add another set of pre-/post-conditions for the `walk` method and verify it again. In the `meow` method of the `Cat` class (lines 21-33 in Figure 3.6), there is a single spec, which requires `this` to have the shape of an object of type `Cat` and the prototype `Cat.prototype`. This is in accordance to the fact that *descendants*$^+$(`Cat`) only contains the element `Cat`. It is important to notice that for all these specifications, the type of `this` does not change.

Constructor specifications. The two specifications corresponding to the constructor of the `Animal` class correspond to the case when we call `new Animal(num)` and when we call the constructor via the `Cat` constructor; in the latter case, the prototype is set to `Cat.prototype`. Regardless of the prototype, the constructor expects to receive an empty `this` object and populates it with the fields that are expected to be found in an instance of `Animal`. The specification of the `Cat` constructor only describes the case when we call `new Cat(num, str)`, since `Cat` is not extended by any class.

The `_super` resource. In the closure where we define `Cat` (Figure 3.6), the `_super` parameter is bound to the `Animal` constructor. In line 17, the `Cat` constructor makes a call to the `Animal` constructor via the `_super` parameter. To verify that call, we need to know that `_super` is indeed bound to the `Animal` constructor in the scope chain of the constructor. By accessing the scope chain via the `CatProtoAndConstructor` predicate, JaVerT is able to verify the binding.

3.3.3 Assignment assertions

Once a variable `x` is declared to have a type τ, it is commonly required that `x` satisfies type τ at all points during the execution of the program. We believe that this is in accordance to the TypeScript programmer's intuition and chose to maintain variable types invariant. A different approach would be requiring that the type invariant holds at the beginning of a unit of execution (e.g. a function) and at the end of it, but can be broken in between.

These different approaches towards the typing invariant lead to different needs in terms of placing assertions. Since we require the invariant to hold at all times, we assert that

```
var Animal = /** @class */ (function () {
  /*
    @id Animal_constructor

    @pre JSObjWithProto(this, #Animalproto) * empty_fields(this : -{ }-) *
      ($$scope == #sc) * (position == #position) * types(#position: Num) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)
    @post Animal(this, #Animalproto) * types(#position: Num) *
      (ret == #ret) * types(#ret: Undef) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)

    @pre JSObjWithProto(this, #Catproto) * empty_fields(this : -{ }-) *
      ($$scope == #sc) * (position == #position) * types(#position: Num) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)
    @post Animal(this, #Catproto) *
      types(#position: Num) * (ret == #ret) * types(#ret: Undef) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)
  */
  function Animal(position) { this.position = position; }
  /*
    @id Animal_walk

    @pre Animal(this, #Animalproto) * ($$scope == #sc) *
      (distance == #distance) * types(#distance: Num) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)
    @post Animal(this, #Animalproto) * types(#distance: Num) *
      (ret == #ret) * types(#ret: Undef) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope) *

    @pre Cat(this, #Catproto) * (distance == #distance) *
      types(#distance: Num) * ($$scope == #sc) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope)
    @post Cat(this, #Catproto) * types(#distance: Num) *
      (ret == #ret) * types(#ret: Undef) *
      AllProtosAndConstructors(#Animal, #Animalproto, #sc,
        #Cat, #Catproto, #Catscope) *
  */
  Animal.prototype.walk = function (distance) {
    this.position += distance; /* @tactic assert(Cat(this, #Catproto)) */
  };
  return Animal;
}());
```

FIGURE 3.5: The JaVerT code corresponding to the TypeScript example of Figure 2.6; fragment illustrating the translation of the `Animal` class

after every assignment to variable x of type τ, $\texttt{Scope}(\texttt{x}, \#x) * \mathscr{C}_T(\#x, \tau)$ holds. It is important to bear in mind that only checking the type of x after an assignment to x or one of its fields is sufficient due to the disjointness assumption under which we carried out the translation. In case we allowed overlaps, changes in x would impact all objects referencing x and the analysis would become untractable.

The approach that allows the typing invariant to be broken would require fewer assertions and would be more efficient. The precise placement of assertions depends on the

```
var Cat = /** @class */ (function (s) {
  __extends(Cat, s);
  /*
    @id Cat_constructor

    @pre JSObjWithProto(this, #Catproto) * empty_fields(this : -{ }-) *
      ($$scope == #sc) * (position == #position) * types(#position: Num) *
      (name == #name) * types(#name: Str) *
      AllProtosAndConstructors(#Animal, #Animalproto, #Animalscope,
        #Cat, #Catproto, #sc) *
    @post Cat(this, #Catproto) * types(#position: Num) *
      types(#name: Str) * (ret == #ret) * (#ret == this) *
      AllProtosAndConstructors(#Animal, #Animalproto, #Animalscope,
        #Cat, #Catproto, #sc)
  */
  function Cat(position, name) {
      var _this = _super.call(this, position) || this;
      _this.name = name;
      return _this;
  }
  /*
    @id Cat_meow

    @pre Cat(this, #Catproto) * ($$scope == #sc) *
      AllProtosAndConstructors(#Animal, #Animalproto, #Animalscope,
        #Cat, #Catproto, #sc) *
    @post Cat(this, #Catproto) * (ret == #ret) * types(#ret: Str) *
      AllProtosAndConstructors(#Animal, #Animalproto, #Animalscope,
        #Cat, #Catproto, #sc) *
  */
  Cat.prototype.meow = function () {
    return "Meow! I'm " + this.name + "!";
  };
  return Cat;
}(Animal));
```

FIGURE 3.6: The JaVerT code corresponding to the TypeScript example of Figure 2.6; fragment illustrating the translation of the `Cat` class

defined unit of execution: if we expect the types to only hold at the beginning and at the end of a function, the function pre-/post-conditions suffice.

The only assertion that we place to check the validity of an assignment is on line 46 in Figure 3.5. We do not check assignments for the `this` object within constructors: the `this` object does not yet have the expected shape and assertions would fail.

3.3.4 Loop invariant assertions

We expect that before entering a loop, all the variables in the current scope satisfy their declared types; this requirement is captured by the $\mathscr{C}_{TE}$ compiler. Since we consider types to be invariant, the same assertion must hold after every iteration of the loop.

3.4 Soundness

We prove the soundness of the translation from TypeScript type annotations to JaVerT assertions. We rely on the guarantees provided by JaVerT (§2.2.4), namely: (1) The

JS-2-JSIL compiler is logic preserving and thoroughly tested; (2) The JS-2-JSIL logic translator is proven correct; (3) JSIL Logic is sound with respect to its operational semantics. These guarantees enable us to reach a soundness result without the need to reason about the subject reduction at the level of JavaScript operational semantics.

The JS Logic satisfiability relation. We define satisfiability for JS Logic assertions with respect to *abstract heaps*, which differs from concrete heaps in that they may map object properties to the special value $\varnothing$. The satisfiability relation for JS Logic assertions has the form: $H, L, l_t, \epsilon \models P$, where: (1) H is an abstract heap; (2) L is the current scope chain; (3) l_t is the binding of the `this` object (4) and ϵ is a JS logical environment, mapping logical variables to values. The satisfiability relation for JS Logic assertions builds on the semantics of JS logical expressions. A logical expression E is interpreted with respect to L, l_t and ϵ, written $[\![E]\!]^{\epsilon}_{L,l_t}$. Both the satisfiability relation and the expression interpretation are mostly standard; we show a fragment below.[1] We make use of the following auxiliary functions:

1. TypeOf, which given a value, outputs its type;
2. *Scope* which given a heap H, a scope chain L and an identifier `x` returns the environment record in which `x` is resolved;
3. *Object* which takes a location as a parameter and returns the partial heap containing an object at the given location;
4. *Function* which takes a location as a parameter and returns the partial heap containing a function object at the given location;
5. *Pi* which takes a heap H, an object location l and a property name `n` and returns the location d of the data descriptor that `n` resolves to in the prototype chain of the object at location l;
6. *DescVal* which, given a heap and the location of a data descriptor, returns its value.

Interpretation of JS Logic Expressions and Satisfiability Relation for Assertions (fragment)

Semantics of Logical Expressions: $[\![\texttt{sc}]\!]^{\epsilon}_{L,l_t} \triangleq L \qquad [\![\texttt{this}]\!]^{\epsilon}_{L,l_t} \triangleq l_t$
$\qquad [\![V]\!]^{\epsilon}_{L,l_t} \triangleq V \qquad [\![\mathrm{x}]\!]^{\epsilon}_{L,l_t} \triangleq \epsilon(\mathrm{x})$

Semantics of GetValue: $GetValue(H, l_{er}, x) \triangleq \begin{cases} H(l_{er}, x) & \text{if } l_{er} \neq lg \\ DescVal(H(l_{er}, x)) & \text{if } l_{er} = lg \end{cases}$

Satisfiability Relation:

$$
\begin{array}{ll}
H, L, l_t, \epsilon \models \mathsf{emp} & \Leftrightarrow H = \mathsf{emp} \\
H, L, l_t, \epsilon \models \mathsf{emptyFields}(E_1 \mid E_2) & \Leftrightarrow H = \biguplus_{m \notin \{[\![E_2]\!]^{\epsilon}_{L,l_t}\}}(([\![E_1]\!]^{\epsilon}_{L,l_t}, m) \mapsto \varnothing) \\
H, L, l_t, \epsilon \models \mathsf{types}(X_i : \tau_i|_{i=1}^{n}) & \Leftrightarrow H = \mathsf{emp} \land \forall i \in \{1, ..., n\}\left[\mathsf{TypeOf}([\![E]\!]^{\epsilon}_{L,l_t}) = \tau_i\right] \\
H, L, l_t, \epsilon \models (E_1, E_2) \mapsto E_3 & \Leftrightarrow H = ([\![E_1]\!]^{\epsilon}_{L,l_t}, [\![E_2]\!]^{\epsilon}_{L,l_t}) \mapsto [\![E_3]\!]^{\epsilon}_{L,l_t} \\
H, L, l_t, \epsilon \models \texttt{Scope}(\mathtt{x}, E) & \Leftrightarrow \exists l_{er}, v \Big[Scope(H, L, \mathtt{x}) = l_{er} \land \\
 & \qquad GetValue(H, l_{er}, \mathtt{x}) = v \land [\![E]\!]^{\epsilon}_{L,l_t} = v \Big] \\
H, L, l_t, \epsilon \models \texttt{Pi}(E_1, E_2, E_3, \_, \_) & \Leftrightarrow Pi(H, [\![E_1]\!]^{\epsilon}_{L,l_t}, [\![E_2]\!]^{\epsilon}_{L,l_t}) = [\![E_3]\!]^{\epsilon}_{L,l_t} \\
H, L, l_t, \epsilon \models \texttt{JSFunctionObject}(E, \_, \_) & \Leftrightarrow H = Function([\![E]\!]^{\epsilon}_{L,l_t}) \\
H, L, l_t, \epsilon \models E_1 = E_2 & \Leftrightarrow [\![E_1]\!]^{\epsilon}_{L,l_t} = [\![E_2]\!]^{\epsilon}_{L,l_t} \\
H, L, l_t, \epsilon \models P * Q & \Leftrightarrow \exists H_1, H_2 \Big[H = H_1 \uplus H_2 \land H_1, L, l_t, \epsilon \models P \land \\
 & \qquad H_2, L, l_t, \epsilon \models Q \Big]
\end{array}
$$

[1] Adapted from [17]

We introduce a typing judgement $H, L, l_t \models \Gamma$ meaning that for all $(\mathtt{x} : \tau) \in \Gamma$, the value x resolves to under H, the scope chain L, given the binding l_t of the `this` object, satisfies type τ. We define an auxiliary typing judgement $H, l_t, v \models \tau$ stating that under the heap H, with the binding l_t of the `this` object, the value v satisfies type τ.

Satisfiability Relation for Types and Typing environments (fragment)

Typing environment satisfiability:

$$H, L, l_t \models (\mathtt{x} : \tau) \uplus \Gamma' \quad \Leftrightarrow \exists H_1, H_2, l_{er}, v \Big[H = H_1 \uplus H_2 \land Scope(H, L, \mathtt{x}) = l_{er} \land$$
$$GetValue(H, l_{er}, \mathtt{x}) = v \land H_1, l_t, v \models \tau \land H_2, L, l_t \models \Gamma' \Big]$$
$$H, L, l_t \models [] \quad \Leftrightarrow H = \mathsf{emp}$$

Type satisfiability:

$$H, l_t, v \models \mathtt{number} \quad \Leftrightarrow \mathsf{TypeOf}(v) = \mathsf{Num}$$
$$H, l_t, v \models \mathtt{string} \quad \Leftrightarrow \mathsf{TypeOf}(v) = \mathsf{Str}$$
$$H, l_t, v \models \mathtt{undefined} \quad \Leftrightarrow \mathsf{TypeOf}(v) = \mathsf{Undef}$$
$$H, l_t, v \models \{\ \mathtt{n}_i : \tau_i|_{i=0}^{n}\ \} \quad \Leftrightarrow \exists H_{obj}, H_0, \dots H_n \Big[H = H_{obj} \uplus \biguplus_{i=0}^{n} H_i \land H_{obj} = Object(v)$$
$$\forall i \in \{0, \cdots, n\} \exists d_i, v_i, H_i', H_i'' \big[$$
$$Pi(H_i', v, \mathtt{n}_i) = d_i \land DescVal(H_i', d_i) = v_i \land$$
$$H_i'', l_t, v_i \models \tau_i \big] \Big]$$
$$H, l_t, v \models \{\ (\mathtt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r\ \} \quad \Leftrightarrow H = Function(v, n)$$

Theorem 1 *For any type environment Γ, if $H, L, l_t, \epsilon \models \mathscr{C}_{TE}(\Gamma)$, then $H, L, l_t \models \Gamma$.*

PROOF SKETCH: We show that if $H, L, l_t, \epsilon \models \mathscr{C}_{TE}(\Gamma)$, then $H, L, l_t \models \Gamma$ by induction on the size of the domain of Γ.

1. CASE: $\Gamma = []$
   PROVE: If $H, L, l_t, \epsilon \models \mathscr{C}_{TE}([])$, then $H, L, l_t \models []$
   1.1. ASSUME: $H, L, l_t, \epsilon \models \mathscr{C}_{TE}([])$
   1.2. $\mathscr{C}_{TE}([]) = \mathsf{emp}$
   PROOF: By def. of $\mathscr{C}_{TE}$.
   1.3. $H, L, l_t, \epsilon \models \mathsf{emp}$
   PROOF: By 1.1 and 1.2.
   1.4. $H = \mathsf{emp}$
   PROOF: By 1.3 and the def. of satisfiability relation for assertions.
   1.5. $H, L, l_t \models [] \Leftrightarrow H = \mathsf{emp}$
   PROOF: By def. of satisfiability relation for typing environments.
   1.6. $H, L, l_t \models []$
   PROOF: By 1.4 and 1.5.
2. CASE: $\Gamma = (\mathtt{x} : \tau) \uplus \Gamma'$
   INDUCTIVE HYPOTHESIS: For any heap H, if $H, L, l_t, \epsilon \models \mathscr{C}_{TE}(\Gamma')$, then $H, L, l_t \models \Gamma'$
   PROVE: If $H, L, l_t, \epsilon \models \mathscr{C}_{TE}((\mathtt{x} : \tau) \uplus \Gamma')$ then $H, L, l_t \models (\mathtt{x} : \tau) \uplus \Gamma'$
   2.1. ASSUME: $H, L, l_t, \epsilon \models \mathscr{C}_{TE}((\mathtt{x} : \tau) \uplus \Gamma')$
   2.2. $H, L, l_t, \epsilon \models \mathtt{Scope}(\mathtt{x}, \#x) * \mathscr{C}_T(\#x, \tau) * \mathscr{C}_{TE}(\Gamma')$
   PROOF: By 2.1 and def. of $\mathscr{C}_{TE}$.
   2.3. There exist H_1 and H_2 such that:
      1. $H = H_1 \uplus H_2$
      2. $H_1, L, l_t, \epsilon \models \mathtt{Scope}(\mathtt{x}, \#x) * \mathscr{C}_T(\#x, \tau)$
      3. $H_2, L, l_t, \epsilon \models \mathscr{C}_{TE}(\Gamma')$
   PROOF: By 2.2 and def. of satisfiability relation for $P * Q$ assertions.
   2.4. $H_2, L, l_t \models \Gamma'$
   PROOF: From conjunct 3 in 2.3 and the inductive hypothesis.

2.5. There exist l_{er} and v, such that:
   1. $Scope(H_1, L, \texttt{x}) = l_{er}$
   2. $GetValue(H_1, l_{er}, \texttt{x}) = v$
   3. $\epsilon(\#x) = v$
   4. $H_1, L, l_t, \epsilon \models \mathscr{C}_T(v, \tau)$

   PROOF: Conjuncts 1, 2, and 3, follow from conjunct 2 in 2.3 and by the satisfiability relation of the Scope predicate. Since Scope is a pure assertion (§2.2.2), we do not need to split H_1 into two disjoint heaps. Conjunct 4 is carried over.

2.6. $H_1, l_t, \epsilon, v \models \tau$

   PROOF: By Theorem 2 and conjunct 4 in 2.5.

2.7. $H, L, l_t \models (\texttt{x} : \tau) \uplus \Gamma'$

   PROOF: By conjunct 1 of 2.3, 2.4, conjuncts 1, 2 of 2.5, 2.6 and the def. of the satisfiability relation for typing environments.

Theorem 2 *For any type environment Γ, value v and type τ, if $H, L, l_t, \epsilon \models \mathscr{C}_T(v, \tau)$, then $H, l_t, \epsilon, v \models \tau$.*

PROOF SKETCH: We show that if $H, L, l_t, \epsilon \models \mathscr{C}_T(v, \tau)$, then $H, l_t, \epsilon, v \models \tau$, by structural induction on the structure of types.

1. CASE: $\tau = \texttt{number}$

   1.1. ASSUME: $H, L, l_t, \epsilon \models \mathscr{C}_T(v, \texttt{number})$

   1.2. $\mathscr{C}_T(v, \texttt{number}) = \mathsf{types}(v : \mathrm{Num})$

   PROOF: By def. of $\mathscr{C}_T$.

   1.3. $H = \mathsf{emp} \wedge \mathsf{TypeOf}(v) = \mathrm{Num}$

   PROOF: By 1.1, 1.2 and the def. of the satisfiability relation for assertions.

   1.4. $H, l_t, \epsilon, v \models \texttt{number} \Leftrightarrow \mathsf{TypeOf}(v) = \mathrm{Num}$

   PROOF: By def. of the satisfiability relation for types.

   1.5. $H, l_t, \epsilon, v \models \texttt{number}$

   PROOF: By 1.3 and 1.4.

2. CASE: $\tau = \texttt{string}$

   Analogous to the case when $\tau = \texttt{number}$.

3. CASE: $\tau = \texttt{undefined}$

   Analogous to the case when $\tau = \texttt{number}$.

4. CASE: $\tau = \texttt{\{ } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r \texttt{ \}}$

   4.1. ASSUME: $H, L, l_t, \epsilon \models \mathscr{C}_T(v, \texttt{\{ } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r \texttt{ \}})$

   4.2. $\mathscr{C}_T(v, \texttt{\{ } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r \texttt{ \}}) = \texttt{JSFunctionObject}(v, \_, \_)$

   PROOF: By def. of $\mathscr{C}_T$.

   4.3. $H = Function(v)$

   PROOF: By 4.1, 4.2, and the satisfiability relation for assertions.

   4.4. $H, l_t, \epsilon, v \models \texttt{\{ } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r \texttt{ \}} \Leftrightarrow H = Function(v)$

   PROOF: By def. of satisfiability relation for types.

   4.5. $H, l_t, \epsilon, v \models \texttt{\{ } (\texttt{x}_i : \tau_i|_{i=0}^{n}) : \tau_r \texttt{ \}}$

   PROOF: By 4.3 and 4.4.

5. CASE: $\tau = \texttt{\{ } \texttt{n}_i : \tau_i|_{i=0}^{n} \texttt{ \}}$

   INDUCTIVE HYPOTHESIS For any $i \in \{0, \ldots, n\}$ and any v_i, if $H, L, l_t, \epsilon \models \mathscr{C}_T(v_i, \tau_i)$, then $H, l_t, \epsilon, v_i \models \tau_i$.

   5.1. ASSUME: $H, L, l_t, \epsilon \models \mathscr{C}_T(v, \texttt{\{ } \texttt{n}_i : \tau_i|_{i=0}^{n} \texttt{ \}})$

   5.2. $\mathscr{C}_T(v, \texttt{\{ } \texttt{n}_i : \tau_i|_{i=0}^{n} \texttt{ \}}) = \texttt{JSObjectGen}(v, \_, \_, \_) * \circledast_{i=0}^{n} \texttt{Pi}(v, \texttt{n}_i, \#desc_i, \_, \_) * \texttt{DescVal}(\#desc_i, \#v_i) * \mathscr{C}_T(\#v_i, \tau_i)$

   PROOF: By def. of $\mathscr{C}_T$.

   5.3. There exist H_i, d_i and v_i for all $i = \{0, \ldots, n\}$ and H_{obj} such that:

1. $H = H_{obj} \uplus \biguplus_{i=0}^{n} H_i$
2. $H_{obj}, L, l_t, \epsilon \models \texttt{JSObjectGen}(v, \_, \_, \_)$
3. $H_i, L, l_t, \epsilon \models \texttt{Pi}(v, \texttt{n}_i, d_i, \_, \_) * \texttt{DescVal}(d_i, v_i) * \mathscr{C}_T(v_i, \tau_i)$
4. $\epsilon(\#desc_i) = d_i$
5. $\epsilon(\#v_i) = v_i$

PROOF: From 5.2, by the satisfiability relation for the $P * Q$ assertion.

5.4. PICK any $i \in \{0, \ldots, n\}$. We have:
$H_i, L, l_t, \epsilon \models \texttt{Pi}(v, \texttt{n}_i, d_i, \_, \_) * \texttt{DescVal}(d_i, v_i) * \mathscr{C}_T(v_i, \tau_i)$
PROOF: Follows directly from 5.3, conjunct 2.

5.5. There exist H_i' and H_i'', such that:
1. $H_i = H_i' \uplus H_i''$
2. $H_i', L, l_t, \epsilon \models \texttt{Pi}(v, \texttt{n}_i, d_i, \_, \_) * \texttt{DescVal}(d_i, v_i)$
3. $H_i'', L, l_t, \epsilon \models \mathscr{C}_T(v_i, \tau_i)$

PROOF: From 5.4, by the satisfiability relation for the $P * Q$ assertion.

5.6. $H_i'', L, l_t, v_i \models \tau_i$
PROOF: From conjunct 3 in 5.5 and the inductive hypothesis.

5.7. $Pi(H_i', v, \texttt{n}_i) = d_i \wedge DescVal(H_i', d_i) = v_i$
PROOF: From conjunct 2 in 5.5, the satisfiability relation for the `Pi` predicate and the fact that `DescVal` is a pure assertion equivalent to the *DescVal* function.

5.8. $\forall i \in \{0, \ldots, n\}$ there exist H_i' and H_i'', such that:
1. $H_i = H_i' \uplus H_i''$
2. $Pi(H_i', v, \texttt{n}_i) = d_i \wedge DescVal(H_i', d_i) = v_i$
3. $H_i'', L, l_t, v_i \models \tau_i$

PROOF: Follows due to the fact i was chosen arbitrarily and from conjunct 1 in 5.5, 5.6 and 5.7.

5.9. $H_{obj} = Object(v)$
PROOF: By satisfiability relation for the `JSObjectGen` predicate and conjunct 2 of 5.3.

5.10. $H, L, l_t, v \models \texttt{\{ } \texttt{n}_i : \tau_i|_{i=0}^{n} \texttt{ \}}$
PROOF:By satisfiability relation for the $\texttt{\{ } \texttt{n}_i : \tau_i|_{i=0}^{n} \texttt{ \}}$ type, 5.8 and 5.9.

3.5 Implementation

The translation presented in §3 is accompanied by an implementation.[2] The tool is easy to use via the command-line: it takes a single required argument, the TypeScript file to be compiled. The output consists of a JavaScript file with the same name as the TypeScript one in case the compilation is successful or an error message in case of failure.

The compiler structure is presented in Figure 3.7. All the possible outcomes of running our compiler are represented in the figure. Invalid TypeScript files are automatically rejected in the first stage of the compilation process. In the second stage, we filter out the programs that do not respect the syntax we presented in §2.1.1. All TypeScript programs which pass the first two stages are expected to compile successfully to a JavaScript file annotated with JaVerT assertions.

As mentioned previously, the translation process consists of two parts: 1. translating the TypeScript code to valid JavaScript ES5 code, handled by `tsc`; 2. translating the TypeScript type annotations to JaVerT assertions and placing them in their respective positions. It is apparent from Figure 3.7 that these two tasks are tightly coupled: the

[2]https://github.com/RaduSzasz/TS2JaVerT

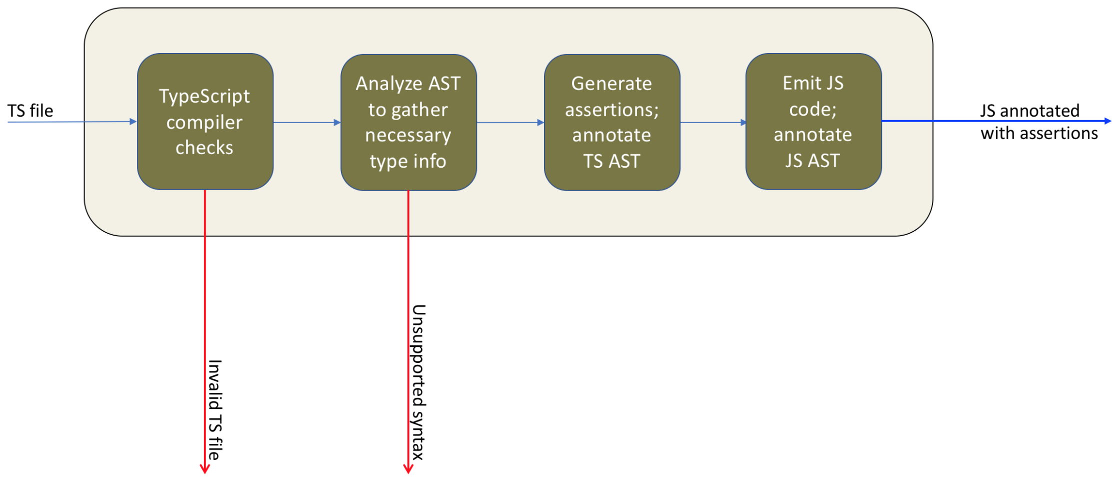
TS file
TypeScript compiler checks
Analyze AST to gather necessary type info
Generate assertions; annotate TS AST
Emit JS code; annotate JS AST
JS annotated with assertions
Invalid TS file
Unsupported syntax

FIGURE 3.7: The structure of the compiler

JaVerT assertions are associated with nodes in the TypeScript AST before the JavaScript is emitted. This means that we do not need to have a mapping between statements in the TypeScript input and statements in the JavaScript output. The TypeScript compiler guarantees that during the transpilation from TypeScript to ES5, the comments will remain associated with their corresponding statements. While this aspect might not appear to be a huge gain for most statements, where the TypeScript compiler only removes type annotations to obtain valid JavaScript, it is vital in the case of classes where the translation process is not as straightforward.

The technology stack. It is apparent from the structure presented in Figure 3.7 that our compiler makes extensive use of features provided by the TypeScript compiler. Requiring access to the TypeScript AST and numerous other features of the TypeScript compiler API (§2.1.4) motivated our choice of technology stack: the compiler is written entirely in TypeScript and only contains few utility libraries as dependencies.

We divide the compilation in four steps, as shown in Figure 3.7:

1. **TypeScript compiler checks.** This step is delegated to the TypeScript compiler, `tsc`. If `tsc` generates any errors, they will be displayed to the user.
2. **Gathering type information.** To generate JaVerT assertions we must firstly possess three pieces of information: (1) the definitions of every user-defined type in the input program; (2) the declared types of every variable and function defined in the input program; (3) the variables which have been assigned to in every statement. We obtain this information in this exact order, by traversing the AST using the visitor pattern [18]. A different visitor is used for each one of these different tasks. The visitors throw an error if the TypeScript program taken as input does not respect the syntax in §2.1.1.
3. **Placing the assertions.** The TypeScript compiler API allows users to manipulate the AST. More specifically, we are able to insert comments before or after any node of the AST. All comments on a certain node must be added at the same time. This restriction is a result of our organisation of the data: we map the type information to the corresponding AST node; after a node is modified, a new one is generated and the mapping is no longer valid. To address this issue, we place the comments representing function specifications, assignment assertions, and loop invariants

using a single visitor. The predicate declarations are placed independently as the first token in the file.

4. **Emitting JavaScript code.** Generating JavaScript ES5 code is delegated to the TypeScript compiler. At the end of this process, we gain access to the AST of the emitted JavaScript. While we should not need to alter the emitted AST in any way, a bug in the TypeScript compiler forces us to do so: when it comes to classes, the TypeScript compiler does not handle synthetic comments placed via the compiler API gracefully—the comments associated with the constructor and initialised instance fields are not displayed. This behaviour is thoroughly documented as an issue in the official TypeScript repository.[3] We use a workaround employed in the Tsickle codebase [20] which involves altering the emitted JavaScript AST. This issue caused significant delays in the development process.

Translating types. The translation from types to assertions is not represented as a standalone step in the diagram in Figure 3.7. The translation is carried out in steps three and four, immediately before the AST nodes are annotated. Once the bug in the TypeScript compiler has been fixed, we will be able to generate and place all assertions at once.

[3]https://github.com/Microsoft/TypeScript/issues/17594

Chapter 4

Evaluation

The current work provides a safe, easy-to-use interface for writing JaVerT specifications for JavaScript programs. We achieved a tremendous reduction in program size via this interface, while ensuring prototype safety and the ability to reason about scope. Our tool provides semi-automatic specification generation: we process TypeScript programs, interpreting types under a disjointness assumption, and generate JavaScript annotated with JaVerT assertions; we then rely on the programmers to enrich the generated assertions using their deep knowledge of the code.

This project is a stepping stone in establishing the relation in between types and separation logic. We evaluate its results from a theoretical and practical standpoint, as stated in the introduction. We then discuss the known limitations and the challenges faced.

4.1 Theoretical results

In §3, we created a translation from TypeScript type annotations to JaVerT assertions, assuming that each object is self-contained, disjoint from all other objects defined. The translation covers a very large subset of the TypeScript types, including object type literals with or without indexing signatures, interfaces, classes, and union types. The only major unsupported types are generic types and arrays; we discuss this decision in §4.3. We prove our translation from TypeScript types to JaVerT assertion to be sound.

We provide a new interpretation for classes, in accordance with the intuition of programmers coming from an OO-programming background: we ensure a certain prototype chain structure and guarantee prototype safety. These two characteristics of our translation of class types lead to more predictable behaviour. We incorporate scope chains into our reasoning, capturing the scope chain at the time of class declaration.

4.2 Practical results

We developed an implementation[1] for the translation from TypeScript to JaVerT assertions. The compiler processes TypeScript files that adhere to the syntax presented in §2.1.1 and outputs JavaScript files that are ready to be analysed by JaVerT. The output specifications are generated by leveraging on the information extracted from the typing environment and they can be verified *automatically* by JaVerT. The programmer can choose to refine these assertions to provide tighter specifications for their code. If they are only interested in checking assertions corresponding to types and that their implementation adheres to our interpretation of types, no additional assertions are required.

[1]https://github.com/RaduSzasz/TS2JaVerT

Example name	Chars (types)	Chars (gen. assertions)	Chars (manual assertions)
ReportFlow	101	1179	0
ReportDynamic	81	2169	224
JaVerTKVMap	81	2169 (3547)	224
JaVerTIdGen	59	1945 (930)	0
JaVerTBST	89	3082 (2050)	0
JaVerTPQ	119	4233 (3705)	0
JaVerTSort	66	2447 (1250)	0
SimpleExtendingClasses	59	3725	0
ComplicatedExtendingClasses	89	7178	0

FIGURE 4.1: Character count of type annotations, JaVerT assertions generated based on the type annotations, and the assertions that needed to be added manually for the program to pass verification

We evaluate our implementation using two metrics:

1. **Ease of code specification.** We measure the difference between the size of TypeScript type annotations and the size of the JaVerT assertions generated. We chose this method over other metrics that are harder to quantify, but are perhaps more meaningful, such as the time spent to write these specifications, especially considering the debugging time that is inherent to manually specifying the code.

2. **Performance.** We mentioned previously that the translation we laid out suffers from a state explosion problem due to subclassing, union types and optional fields. We evaluate the time required to generate the specifications and the time required to verify the generated code using JaVerT.

4.2.1 Ease of code specification

We measure the difference in character count achieved by using our tool instead of hand-writing specifications. The character count for generated assertions includes the predicate definitions as well as the assertions placed within the code. The character count for type annotations includes the annotations specifying variable types, parameter types, function return types, and interface types. Most notably, the `class` syntax does not contribute towards the character count, as it is equivalent to the manual setting of the prototype chain structure in the emitted ES5 code.

We present the results in Figure 4.1. The name of the example is the one that is used in the repository. The examples come from different sources:

1. The ReportFlow and ReportDynamic examples correspond to the two code snippets which lead to unsoundness in TypeScript: Figure 1.3 and Figure 1.4. As indicated in the motivation, these two examples are expected to fail verification.

2. Other examples from the report: we ran our compiler on the TypeScript code presented in Figure 2.4b, Figure 2.6, and Figure 3.1 and achieved the expected result in all cases. For the code in Figure 2.4b and Figure 3.1 the verification failed, as expected. We successfully verified a version of the TypeScript code in Figure 2.6 augmented with two statements, creating a `Cat` and making it `meow`.

3. The examples whose name starts with JaVerT are adapted from the JaVerT repository: an id generator, a key-value map, a binary search tree, a priority queue, and an implementation of an insertion sort. We translated the single-file JavaScript

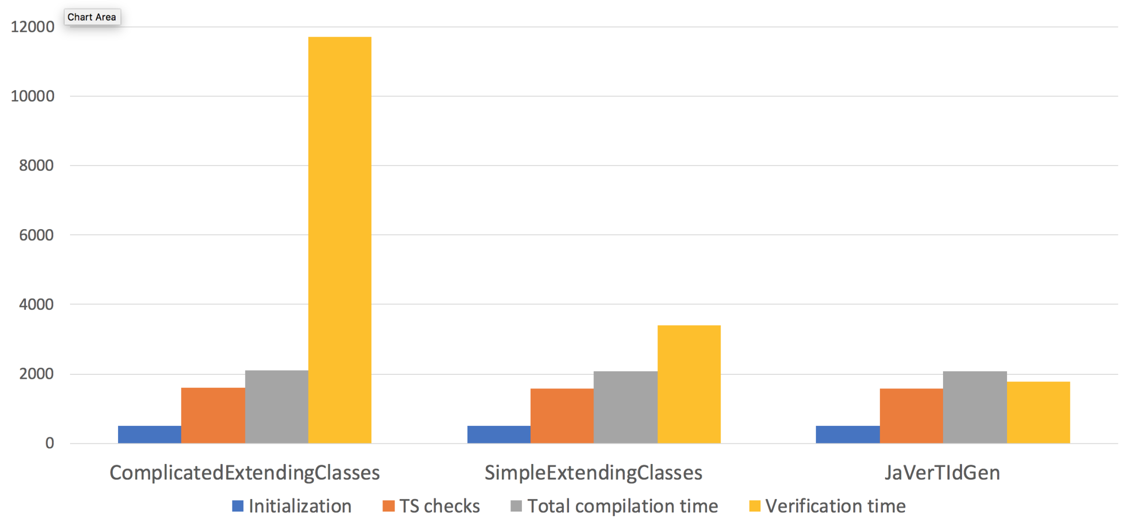

FIGURE 4.2: Time spent compiling/verifying three examples.

implementations from the JaVerT repository to TypeScript, making use of classes where these would fit the semantics of the code. We provide the character count for the original specifications in the JaVerT repository in parenthesis.

Using type annotations to specify programs leads to much shorter specifications: for most programs, roughly 40 times less characters are needed to write type annotations than JaVerT assertions. The best ratio is offered by the ComplicatedExtendingClasses example which requires 89 characters to be annotated with types, while the generated assertions sum up to 7178 characters; the character count for type annotations represents only 1.2% of the character count for assertions. At the opposite end of the spectrum is ReportFlow: it requires 101 characters of type annotations to generate 1179 characters of assertions; the character count for type annotations represents only 8.5% of the character count for assertions.

For the JaVerT examples, we included the character count used in the JaVerT repository. The original specifications are shorter, with the exception of the key-value map example. This is due to the simpler structure these examples employ in the original JavaScript code; our use of classes makes the code more structured, but complicates the specifications.

4.2.2 Performance

We assess the capacity of the toolchain to be used in a production environment. To this end, we measure the type spent translating the TypeScript files to JavaScript code annotated with JaVerT assertions, as well as the time spent verifying the files.

We used the built-in `node` [16] functionality to measure the time the compiler spent during the different stages of the translation and the `time` command-line utility to measure the verification time. The measurements are made on a machine with an Intel Core i7-4770HQ 2.2GHz CPU and 16GB 1600 MHz DDR3 RAM; for each file presented, the measurements are averaged over ten runs carried out under similar conditions: idle machine, fully charged, charger connected. We did not consider external factors such as ambient temperature in the measurements.

We chose three examples of various size and complexity that we considered to be representative of different file types that might be encountered in practice. The results are presented in Figure 4.2.

The time for compiling each of the examples is around 2.1 seconds. The execution time is dominated by the initialisation and the checks performed by `tsc`. The time spent gathering type information is around 8ms, while annotating the AST takes an additional 10ms, on average. These results show that the translation phase would not affect the user in any meaningful way. The overall time needed to run the compiler could be drastically reduced by introducing a *watch* mode, which tracks changes in files, rather than recompiling them from scratch every time. This interface is exposed by the TypeScript compiler API and hence, we would be able to incorporate the *watch* functionality in our compiler.

The burden in the process, hence, lies in the time spent verifying the produced specifications using JaVerT. The JaVerT verification times strongly depend on the contents of the program being verified: several type annotations cause an exponential increase in the number of specifications that require verification; a complicated logic with many branchings or loops also leads to increased verification times. For the JaVerTIdGen example, using a single class, the verification time is only 1.8 seconds. In examples with more complex class hierarchies, the number of specifications that need to be verified grows: for SimpleExtendingClasses[2], JaVerT needs 3.4 seconds to complete verification; for the ComplicatedExtendingClasses example, where we add a new class `Sphynx`, extending `Cat`, verification takes almost 12 seconds. Verification time increases dramatically as we operate on more complex data structures: the binary search tree takes over four minutes to verify. JaVerT does not currently support the *watch* functionality and, hence, we cannot expect improvements in verification times for files that have previously been analysed. It is possible for this functionality to be introduced into JaVerT in future releases.

4.3 Known limitations

We discuss several limitations of our tool below.

The disjointness assumption. The main limitation of the tool is the fact that it carries out the translation assuming that every object is self-contained and disjoint from all other objects. This means our tool is unable to reason about aliasing. For example, for a linked list, (Figure 4.3), we are unable to specify the following:

1. the `findElementNode` method—the returned element is an element in the list and this leads to duplicated resource;
2. the `makeCircular` method—by unfolding the `LinkedListNode` predicate a sufficient number of times times, we would get duplicated resource corresponding to `this.head`; and
3. an optimised list—if we added a `tail` node to our `LinkedList` data structure, this would constitute aliasing; our `LinkedList` predicate would be invalid.

State explosion. The current translation causes an explosion in the number of states that need to be verified when three features are used: subclassing, union types, and optional fields. All these type annotations are translated as disjunctions. The number of

[2]The example presented in Figure 2.6.

specifications that need to be checked for any given function grows exponentially with the number of disjunctions, as JaVerT only supports disjunctions at the top level.

Lack of higher-order reasoning. JaVerT lacks the ability to reason about:

1. **Higher-order functions.** JaVerT associates a unique id with the body of every function, which enables pre-/post-conditions to be associated with that function. Hence, in order to be able to call a function during verification, we must know its id; this contributes to the above-mentioned state explosion, as we must create a specification for every possible id.

2. **Generic types.** Our approach requires of us to write assertions describing the shape of the heap, but for generic types we do not possess any information on the potential shape of the object and hence cannot describe the part of the heap in which they are contained. The problem encountered using generics is twofold: **(1)** verifying the implementation of the functions/classes using type variables is difficult; **(2)** client code cannot make use of the fact that it knows what type variables are instantiated to: JaVerT does not allow predicates to be passed as arguments to other predicates and hence, we are unable to perfectly describe the structures we are working with in the case of generic code.

Lack of support for arrays. JaVerT lacks a suitable abstraction for JavaScript arrays. In TypeScript, arrays are generic types and, hence, the fact that JaVerT lacks support for arrays is augmented by the lack of support for generics.

Other missing TypeScript syntax. Besides the aforementioned lack of support for generics and arrays, there are other aspects of TypeScript not supported in the current implementation. For brevity, we omitted some features that would have posed no problems, such as for loops, error throwing, intersection types, and `const`/`let` declarations. We chose to not support: (1) **type guards** [26], since we felt user-defined type checks are prone to errors and would be a potential weak point in our verification process; (2) **iterators** [28], whose support is limited to the `Array` interface when transpiling TypeScript to ES5; (3) **modules and namespaces** [29], which would have complicated our translation from TypeScript to ES5 greatly, introducing challenges orthogonal to the current work.

4.4 Lessons learnt

The current work is different from any of my prior experience: it contains a heavy theoretical component and is a large piece of work which requires sustained effort. Previous tasks encountered in my undergraduate years and as part of my internship experiences were much more well defined, required less exploration, were mostly practical in nature and were much smaller in size. There are two main lessons that I have learnt as part of this project.

Base work on examples. We started this project by building the translation formalism. The practical examples we had at the time were limited to the two motivation examples. These examples did not highlight the extent to which the disjointness assumption impacts the work. Had we explored further examples before starting to formalise the translation, we would have better understood the implications of the disjointness assumption and would have had more time to explore the aspects detailed in §5. Similarly, the development of the formalism was slowed down and required multiple iterations because of cases we did not initially consider; most of the issues encountered had to do with subclassing.

```
class LinkedListNode {
  public val: number;
  public next: LinkedListNode | undefined;
  /** Constructor and potentially other methods **/
}

class LinkedList {
  private head: LinkedListNode | undefined;
  /** Constructor and other methods **/
  public makeCircular(): void {
    var currNode: LinkedListNode | undefined;
    while (currNode) {
      if (currNode.next === undefined) {
        currNode.next = this.head;
        return;
      }
      currNode = currNode.next;
    }
  }
  public findElementNode(val: number): LinkedListNode | undefined {
    var currNode = this.head;
    while (currNode) {
      if (currNode.val === val) {
        return currNode;
      }
      currNode = currNode.next;
    }
  }
}
```

FIGURE 4.3: Linked list implementation in TypeScript

Make it clear when you are uncomfortable. There was a considerable amount of time that did not produce as much impact as it could have because I did not voice the fact that I was uncomfortable with the task at hand and did not ask for clarifications and help. The two concrete examples are delaying asking for support for the soundness proof and help in learning how to debug the traces produced by JaVerT myself. I am confident that for future work I will communicate better and mitigate this issue.

4.5 Comparison with other works

We compare the current work with two distinct approaches: ensuring type safety for JavaScript on the one hand, and verifying properties of programs by checking separation logic specifications on the other hand.

4.5.1 Type systems for JavaScript

The two main attempts to retrofit a type system on top of JavaScript, TypeScript and Flow, provide a smooth onboarding experience for programmers and good performance, suitable for the environment that JavaScript developers are used to operate in. Both of these have holes in their type systems and can lead to potential unsoundness.

Analysing the code with JaVerT produces stronger safety guarantees than either of the other options. Via the current work, JaVerT provides the same easy to use interface as TypeScript. Our approach requires type annotations to be present for every variable declaration, parameter and return value, while TypeScript and Flow feature type inference mechanisms.

An important aspect is that while Flow and TypeScript handle the entire JavaScript language, we are restricted to a subset (§4.3). The type annotations that we support are not as expressive as those in either of these other languages. Hence, the advantage of automatically generated specifications is reduced; the developer needs to fill in the missing information that we are unable to extract from the typing environment. The verification times are also an impediment for the wide adoption of the tool (§4.2.2).

We believe that the current work facilitates programmers to verify their critical JavaScript code, where the lack of soundness of TypeScript and Flow is unacceptable, but we consider the tool to be far from entering the mainstream JavaScript toolchain.

4.5.2 Generating Separation Logic assertions

Other attempts to prove correctness of programs using separation logic assertions, such as jStar and VeriFast, both require the user to write verbose specifications for the methods they define. Our work has the major contribution of offering an expressive interface, enabling users to specify their code without any prior knowledge of separation logic. It is likely that programmers need to familiarise themselves with the underlying assertion language—as shown in §4.3, the assertions we generate are often not illustrating the programmer's intent, but the bulk of the specification is generated automatically, so we can confidently say that the users are more likely to successfully specify their code.

Comparisons with other tools in respects other than ease of specification writing provide little insight. The aim of the current work is to enable efficient specification writing. Metrics such as verification time are relevant for JaVerT in comparison to the tools discussed. We show in §4.2.2 that the time spent generating specification is insignificant compared to the verification time. Expresiveness is another metric where comparisons are meaningless: our compiler only emits assertions that are analogous to type annotations, which only constitute a subset of what hand-written assertions can talk about.

Chapter 5

Conclusion and future work

The current work provides a safe, easy-to-use interface for writing JaVerT specifications for JavaScript programs. We achieved a tremendous reduction in program size via this interface, while ensuring prototype safety and the ability to reason about scope.

On top of that, the current work opened the door towards a multitude of projects. On the one hand, there are opportunities to develop the tool as it is, together with JaVerT, which is a viable option for developers who desire safe JavaScript code. On the other hand, there is ample opportunity to discuss whether the current interpretation of types is in accordance with the programmer's intuition and the potential role separation logic could have in developing new safer type systems.

5.1 Future Work

5.1.1 Tool enhancements

We believe that the tool offers a viable alternative to hand-writing JaVerT specifications, making it easier for programmers to verify their code using JaVerT. We believe some additional feature would reduce friction significantly for users.

1. **Extending the TypeScript type annotations.** TypeScript type annotations do not provide enough information for us to generate tight specifications. We believe there is potential in trying to investigate alternative type annotations or extend the TypeScript ones to provide greater expressivity.
2. **Type inference.** Most TypeScript programmers do not annotate every single variable with its corresponding type and instead allow the TypeScript compiler to infer the type. We could operate on the information inferred by `tsc` since it is made available via the TypeScript compiler API. If the type inference algorithm produced the wrong type, an error would be detected during verification and the programmer could explicitly annotate the given variable/function.
3. **Lifting errors to the level of TypeScript.** JaVerT produces logs that are often longer than a million lines and are hard to debug, even for experienced users. The situation would be far worse for a programmer relying on our tool for specifying their code who is not familiar with JaVerT. Lifting errors from the level of JSIL to that of JavaScript and then to the level of TypeScript constitutes a difficult challenge. Solving it would greatly improve the usability of the entire toolchain.

5.1.2 Separation logic and types

Throughout the current work, we used the separating conjunction to model the assertion corresponding to class types. We showed how this requires us to carry out our

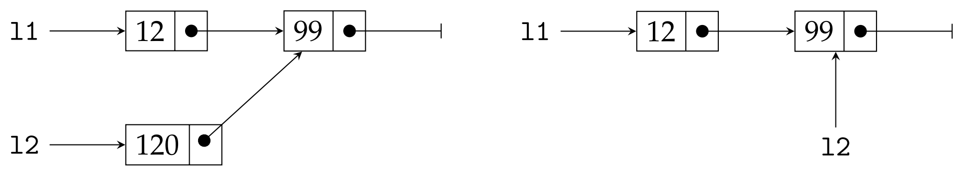

FIGURE 5.1: Possible overlaps for two linked lists

translation under a disjointness assumption that is not in accordance with the TypeScript interpretation of types.

To describe the heap satisfying a typing environment, given the TypeScript interpretation of types, we need to use the classical logic $\wedge$ operator, rather than the $*$ or even the more permissive sepish operator, $\cup\!\!\!\ast$ [19]. However, separation logic can enforce properties that the TypeScript type system cannot:

1. **Structure.** It is common to assume that a linked list, such as that in Figure 4.3, is not circular. However, the TypeScript type system cannot provide that guarantee. With separation logic, we can easily generate a predicate that requires the list to end with a pointer to `null`. Enabling tighter specifications for data structures using separation logic assertions is worth analysing.

2. **Resource ownership.** We consider two linked lists, `l1` and `l2`, satisfying the types presented in Figure 4.3. In Figure 5.1 we present possible overlaps for these two linked lists. Neither `l1` nor `l2` have ownership over the list that they denote, which makes it harder for programmers to ensure invariants within their code. Separation logic is able to reason about this kind of behaviour by modelling disjointness via the $*$ operator. We believe that there is potential in exploring the relation between separation logic and ownership types [8, 34], especially since there exists an intuitive relation between “objects in boxes” [8] and heaplets in separation logic.

We find providing safety guarantees via better descriptions of data structures and resource ownership to be very exciting future work. There are strong indications that using separation logic to reason about types in this new system and JaVerT to check specifications can be successful.

Bibliography

[1] C. Anderson, P. Giannini, and S. Drossopoulou. "Towards Type Inference for JavaScript". In: *ECOOP 2005 - Object-Oriented Programming*. Ed. by A. P. Black. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 428–452.

[2] G. Balakrishnan and T. Reps. "Recency-abstraction for heap-allocated storage". In: *SAS*. Vol. 6. Springer. 2006, pp. 221–239.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. "Intersection and union types: syntax and semantics". In: *Information and Computation* 119.2 (1995), pp. 202–230.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. "Smallfoot: Modular automatic assertion checking with separation logic". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 115–137.

[5] J. Berdine, C. Calcagno, and P. O'Hearn. "Symbolic execution with separation logic". In: *APLAS*. Vol. 5. 3780. Springer. 2005, pp. 52–68.

[6] G. Bierman, M. Abadi, and M. Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*. Ed. by R. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9. DOI: 10.1007/978-3-662-44202-9_11. URL: https://doi.org/10.1007/978-3-662-44202-9_11.

[7] G. Bracha and L. Bak. *Dart, a new programming language for structured web programming*. GOTO Aarhus. 2011.

[8] N. R. Cameron et al. "Multiple Ownership". In: *SIGPLAN Not.* 42.10 (Oct. 2007), pp. 441–460. ISSN: 0362-1340. DOI: 10.1145/1297105.1297060. URL: http://doi.acm.org/10.1145/1297105.1297060.

[9] A. Chaudhuri et al. "Fast and Precise Type Checking for JavaScript". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 48:1–48:30. ISSN: 2475-1421. DOI: 10.1145/3133872. URL: http://doi.acm.org/10.1145/3133872.

[10] G. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[11] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.

[12] D. Distefano and M. J. Parkinson. "jStar: Towards practical verification for Java". In: *ACM Sigplan Notices*. Vol. 43. 10. ACM. 2008, pp. 213–226.

[13] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1. 2011. URL: https://www.ecma-international.org/ecma-262/5.1.

[14] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 6th ed. 2015. URL: https://www.ecma-international.org/ecma-262/6.0.

[15] A. Feldthaus and A. Møller. "Checking Correctness of TypeScript Interfaces for JavaScript Libraries". In: *SIGPLAN Not.* 49.10 (Oct. 2014), pp. 1–16. ISSN: 0362-1340. DOI: 10.1145/2714064.2660215. URL: http://doi.acm.org/10.1145/2714064.2660215.

[16] Node.js Foundation. *Node.js*. 2012-2017. URL: https://github.com/nodejs/node (visited on 05/26/2018).

[17] José Fragoso Santos et al. "JaVerT: JavaScript Verification Toolchain". In: *Proc. ACM Program. Lang.* 2.POPL (Jan. 2018), 50:1–50:33. ISSN: 2475-1421. DOI: 10.1145/3158138. URL: http://doi.acm.org/10.1145/3158138.

[18] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[19] P. Gardner, S. Maffeis, and G. Smith. "Towards a program logic for JavaScript". In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 31–44.

[20] Google. *Tsickle*. https://github.com/Angular/Tsickle/. 2014.

[21] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[22] Facebook Inc. *Flow*. 2014-2018. URL: https://flow.org (visited on 06/16/2018).

[23] Github Inc. *Github website*. 2018. URL: https://octoverse.github.com (visited on 01/17/2018).

[24] B. Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java." In: *NASA Formal Methods* 6617 (2011), pp. 41–55.

[25] S. H. Jensen, A. Møller, and P. Thiemann. "Type Analysis for JavaScript." In: *SAS*. Vol. 9. Springer. 2009, pp. 238–255.

[26] Microsoft. *Advanced Types*. 2012-2017. URL: https://www.typescriptlang.org/docs/handbook/advanced-types.html (visited on 01/21/2018).

[27] Microsoft. *CompilerOptions*. 2012-2018. URL: https://www.typescriptlang.org/docs/handbook/compiler-options.html (visited on 03/29/2018).

[28] Microsoft. *Iterators and Generators*. 2012-2018. URL: https://www.typescriptlang.org/docs/handbook/iterators-and-generators.html (visited on 06/17/2018).

[29] Microsoft. *Namespaces And Modules*. 2012-2018. URL: https://www.typescriptlang.org/docs/handbook/namespaces-and-modules.html (visited on 06/17/2018).

[30] Microsoft. *TypeScript*. https://github.com/Microsoft/TypeScript/. 2012.

[31] Microsoft. *TypeScript*. 2012-2018. URL: https://www.typescriptlang.org (visited on 06/16/2018).

[32] Mozilla and individual contributors. *Defining classes*. 2005-2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes (visited on 01/21/2018).

[33] Mozilla and individual contributors. *Object.prototype*. 2005-2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype (visited on 06/16/2018).

[34] J. Noble, J. Vitek, and J. Potter. "Flexible alias protection". In: *ECOOP'98 — Object-Oriented Programming*. Ed. by E. Jul. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 158–185. ISBN: 978-3-540-69064-1.

[35] P. O'Hearn, J. Reynolds, and H. Yang. "Local reasoning about programs that alter data structures". In: *Computer science logic*. Springer. 2001, pp. 1–19.

[36] M. Parkinson and G. Bierman. "Separation logic and abstraction". In: *ACM SIGPLAN Notices*. Vol. 40. 1. ACM. 2005, pp. 247–258.

[37] M. J. Parkinson and G. Bierman. "Separation logic, abstraction and inheritance". In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 75–86.

[38] A. Rastogi et al. "Safe & efficient gradual typing for TypeScript". In: *ACM SIGPLAN Notices*. Vol. 50. 1. ACM. 2015, pp. 167–180.

[39] J. Reynolds. "Separation logic: A logic for shared mutable data structures". In: *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE. 2002, pp. 55–74.

[40] P. Thiemann. "Towards a type system for analyzing JavaScript programs". In: *ESOP*. Vol. 3444. Springer. 2005, pp. 408–422.