Gradual Typing Embedded Securely in JavaScript

N. Swamy C. Fournet A. Rastogi K. Bhargavan J. Chen P.-Y Strub G. Bierman
MSR INRIA IMDEA

Abstract

JavaScript's flexible semantics and lack of types make writing correct code hard and writing secure code extremely difficult. To address the former problem, various forms of gradual typing have been proposed, for example, Closure and TypeScript. However, supporting all common programming idioms is not easy; for example, the TypeScript type system is deliberately unsound. Here we address also the latter problem, and propose a gradual type system and implementation technique that additionally provides important security guarantees.

We present TS*: a gradually-typed core of JavaScript. Like TypeScript, TS* is translated to JavaScript before execution. In contrast, TS* features full runtime reflection over three kinds of types: (1) simple types for higher-order functions, recursive datatypes and dictionary-based extensible records; (2) the type `any`, for dynamically type-safe TS* expressions; and (3) the type `un`, for untrusted, potentially malicious JavaScript contexts in which TS* is embedded. Our main theorem guarantees the type safety of TS* despite its interactions with arbitrary JavaScript contexts, which are free to use `eval`, stack walks, prototype customizations, and other offensive features. Its proof employs a novel use of type-preserving compilation, wherein we prove all the runtime invariants of the translation of TS* to JavaScript by showing that translated programs are well-typed in JS*, a previously proposed dependently typed language for proving functional correctness of JavaScript programs.

We describe a prototype compiler, a secure runtime, and sample applications for TS*. Our examples illustrate how web security patterns that developers currently program in JavaScript (with much difficulty and still with dubious results) can instead be programmed naturally in TS*, retaining a flavor of idiomatic JavaScript, while providing strong safety guarantees by virtue of typing.

1. Introduction

Writing secure JavaScript is hard. Even simple functions, which appear safe on the surface, can be easily broken. As an illustration, consider the script below, where `protect(send)` returns a function that interposes an access control check on `send`.

```
function send(url,msg) { ... }
function protect(send) {
var whitelist={"http://www.google.com/mail":true,
"http://www.google.com/plus":true};
return function (url, msg) {
if (whitelist[url]) { send(url, msg); }}}
send = protect(send);
```

If this script were to run in isolation, it would achieve its intended functionality. However, JavaScript programs rarely run in isolation—programmers explicitly link their code with third-party frameworks, and, worse, unexpected code fragments can be injected into the web sandbox by cross-site scripting attacks. For example, the following script running in the same sandbox as `protect` could succeed in sending a message to an unintended recipient.

```
Object.prototype["http://evil.com"]=true;
send("http://evil.com", "bypass!");
```

This is just one attack on `protect`, a function simplified from actual scripts in the OWASP CSRF and Facebook APIs (see §6) that intend to protect calls to `XMLHttpRequest` and `PostMessage`, respectively, instead of `send`. Experimentally, we found and reported several security flaws in their scripts, suggesting that such APIs are difficult to reliably protect from arbitrary JavaScript. Unintended callbacks to an untrusted caller (caused, for example, by implicit coercions, getters or setters, prototype hacking, and global object overwriting) are difficult both to prevent and to contain.

1.1 Attacks ≈ Type Errors

Arguably, each of these attacks can be blamed to some uncaught type errors. Almost *any* dynamic type-safety error while running a sensitive script can be exploited by a malicious context, as follows. Anticipating that the script will dereference an otherwise undefined property `x`, which in JavaScript would just return the value `undefined`, a hostile context can define

```
Object.defineProperty(Object.prototype,"x",
{get:function(){/*exploit*/}});
```

then run the script and, as `x` is dereferenced and triggers the callback, access any argument on the caller stack. Thus, for protecting good scripts from bad ones, type errors in JavaScript are just as dangerous as buffer overruns in C. Despite the numerous dynamic checks performed by the JavaScript runtime, some stronger notion of type safety is called for.

Other researchers have made similar observations. For example, at POPL last year, Fournet et al. (2013) show several attacks on a similar piece of code and argue that carefully controlling the interaction between a script and its context is essential for security. They show how to compile F*, a statically typed, ML-like source language to JavaScript in a fully abstract way, allowing programmers to write and reason about functions like `protect` in ML, while their compiler generates secure JavaScript automatically. This is an attractive design, but the sad truth is that millions of JavaScript programmers are unlikely to switch to ML. Meanwhile, Bhargavan et al. (2013) have developed DJS, a minimal, statically typed, secure core of JavaScript, primarily for writing first-order string-processing functions using arrays of fixed size. DJS is suitable for writing security-critical code, like cryptographic libraries, and Bhargavan et al. prove that the behavior of programs accepted by the DJS type checker is independent of the JavaScript environment in which they may run. A third line of work is by Taly et al. (2011), who propose a subset of JavaScript called SES$_{light}$, and a static analysis for it to decide whether or not a program is isolated from its environment. Although we find this a promising line to pursue, isolation in SES$_{light}$ currently works only if the entire environment in which the program runs is also programmed in SES$_{light}$, an assumption hard to validate in the presence of cross-site scripts. A fourth line of work considers dynamic

information flow type systems for JavaScript (Austin and Flanagan 2012; De Groef et al. 2012; Hedin and Sabelfeld 2012), but their guarantees only hold when all the code in the JavaScript context can be typed (at least at the lowest security level).

1.2 TS*: a gradually type-safe language within JavaScript

This paper presents TS*, a source programming language that retains many of the dynamic programming idioms of JavaScript, while ensuring type-safety even in an untrusted JavaScript environment. TS* supports writing functions like `protect` exactly as shown, while a compiler from TS* to JavaScript ensures that the access control check in `protect` cannot be subverted.

Although significantly more flexible than F* or DJS, TS* still rules out many inherently unsafe features of JavaScript for the code we protect, thereby enforcing a stricter programming discipline and facilitating security code reviews. We intend TS* to be used to protect security-critical scripts—guarding, for instance, sensitive resources and capabilities—executed on web pages that also include dynamically loaded, untrusted, potentially malicious scripts. By necessity, most of the code running on these pages is provided by third parties; for the moment, we leave such code unchanged (and unprotected). Nonetheless, by lightly rewriting the security-critical scripts, and gradually typing them, we enforce a strong notion of dynamic type-safety. In addition, we protect the scripts' interface, using type-directed wrappers to shield them from adversarial JavaScript environments. This places these scripts on a robust type-safe foundation, from which we can reason about their security.

TS* is inspired by TypeScript,[1] a recent extension of JavaScript with a type system based on gradual typing (Siek and Taha 2006). TypeScript provides primitive types like `number` and `string`, function and object types, as well as interfaces and classes. The most recent version supports generic types. All types are subtypes of `any`, the type of arbitrary JavaScript code. In less than a year since its release, TypeScript has seen promising uptake by the JavaScript community, e.g., over 150 popular JavaScript frameworks and libraries now have TypeScript definitions.[2] Other gradually typed variants of JavaScript, like Google's Closure,[3] are also commonly used. However, these type systems do not provide any runtime safety guarantees. Indeed, since TypeScript and Closure permit programs to include all the unsafe features of JavaScript, their type systems are intentionally unsound. Even unsound types provide value, e.g., they can be used for automatic code-completion, but sound types can do so much more! Hence, we design and implement TS*, with the following features:

A statically typed core of functions, datatypes and records. The base type system of TS* includes primitive types like `unit`, `bool`, `number` and `string`; higher-order function types, recursive datatypes, and extensible records of fields with optional mutability annotations. Records are equipped with a structural subtyping relation. For example, the type `point` defined below is a subtype of a record that omits some of its fields, and function subtyping is, as usual, contravariant on the arguments and covariant on the results.

```
{x:mutable number;y:mutable number;setX:number->unit}
```

Dynamically typed fragment. We introduce a type `any`, for dynamically typed TS* expressions. All the types from the statically typed core are subtypes of `any`, and any TS* term whose subterms all have type `any` can always be given the type `any`. In the spirit of JavaScript, in the `any`-fragment, we view records as extensible dictionaries with string-valued keys. TS* supports the use of computed properties, e.g., in `any`-typed code, expressions like `p["set"+ "X"]` or `whitelist[url]` are legal ways to safely project the appropriate field from the underlying object, if it exists. As far as we are aware, ours is the first gradual type system to soundly support dictionary-based mutable records and computed properties.

Runtime reflection over types. Case analysis on the runtime type of a value is a common idiom in JavaScript and other dynamically typed languages—Guha et al. (2011) present several typical uses of JavaScript's `typeof` operator. TS* embraces this idiom and compiles programs with runtime type information (RTTI) to support introspection on *all source types* at runtime, e.g., `<isTag point>p` checks if `p`'s RTTI is a structural subtype of `point`. In addition to providing an expressive source programming construct, RTTI also forms the basis of an efficient and simple enforcement mechanism for gradual typing. Interactions between the statically typed core and `any` do not require further complex language features like wrappers for higher-order contracts (Findler and Felleisen 2002).

`un`, the type of the adversary, mediated by wrappers. Finally, and most distinctively, TS* provides a *second dynamic type*, `un`, the type of arbitrary, potentially adversarial JavaScript expressions. Our `un` type is reminiscent of types for adversaries, as proposed by Gordon and Jeffrey (2001). However, unlike prior uses of `un` in the context of secure compilers (e.g. Fournet et al. 2013), `un` is a first-class type in TS*: `un` values may be stored in records, used as arguments and results of functions, etc. However, `un` is incomparable to `any` in the subtyping relation and, in contrast with `any`, all operations on `un` values are mediated by (higher-order) wrappers that safely build coercions to and from `un` (as well as other types). The wrappers enforce a strict heap separation between the `un`-context and TS*, ensuring that adversarial code cannot break the internal invariants of TS*.

1.3 Evaluating TS*: theory and practice

We have developed a prototype implementation of TS*. Our implementation takes as input TS* concrete syntax (which resembles JavaScript with type annotations) and emits JavaScript concrete syntax.

We formalize our compiler as a type-directed translation relation (§3). To formalize properties of the translated program, we give TS* a translation semantics to JS*, a dependently typed model of JavaScript developed by Swamy et al. (2013), which is in turn based on λJS by Guha et al. (2010). Precise monadic refinement types in JS* allow us to conveniently phrase our metatheory (§4) in terms of type-correctness of JS*, yielding three main properties:

- *Memory isolation*: the adversary cannot directly read, write, or tamper with TS* objects.
- *Static safety*: statically typed code is safely compiled without any runtime checks, even in the presence of type-modifying changes to objects.
- *Dynamic safety*: runtime type information is sound and at least as precise as the static type.

Experimentally, we evaluate TS* by programming and adapting several security-sensitive JavaScript web libraries (§6). Our examples include a OWASP reference library to protect against cross-site request forgeries (CSRF) (Barth et al. 2008); and an adaptation of secure login and JSON-validation scripts within the Facebook API.[4] Our main contributions include:

(1) the design of a gradual type system and RTTI support for safely composing statically typed, dynamically typed, and arbitrary JavaScript; (§3)

(2) a type safety theorem and its proof by translation to JS*; (§4)

[1] http://www.typescriptlang.org/

[2] https://github.com/borisyankov/DefinitelyTyped

[3] https://developers.google.com/closure/compiler/

[4] https://developers.facebook.com/docs/reference/javascript/

(3) a prototype implementation, including a protocol to ensure that our runtime support runs first on pages hosting compiled TS⋆, and securely initializes our type invariant; (§5)

(4) security applications, illustrating a series of authorization and access control patterns taken from popular security-sensitive web applications and libraries, motivated by new attacks. (§6)

Our results lead us to conclude that TS⋆ provides a novel, effective application of gradual typing as the foundation of secure programming within JavaScript.

The latest version of our compiler, including programming examples, attacks, and sample web application deployments, is available at http://research.microsoft.com/fstar.

2. An overview of TS⋆

We begin by presenting the design of TS⋆ informally, using several small examples for illustration.

2.1 Gradually securing programs by moving from un to any

While we envisage TS⋆ as the basis of a full-fledged gradually typed web-programming language, we initially consider JavaScript programmers willing to harden safety- and security-critical fragments of their code. They can start by giving their existing JavaScript code the type un in TS⋆, and then gradually migrating selected fragments to safe (but still dynamically typed) any-typed code in TS⋆.

This exercise is valuable since any code in TS⋆ enjoys a *memory isolation* property, a robust foundation upon which to build secure sub-systems of a larger program. Memory isolation alone prevents many common attacks. For example, the prototype poisoning attack of §1 occurs because of a failure of memory isolation: the command `whitelist[url]` causes a prototype-chain traversal that ends with reading a field of `Object.prototype` which, unfortunately, is a reference to an object controlled by the adversary. By re-using `protect`, unchanged, as a TS⋆function, the whitelist has type `any`, and the attack is foiled. Specifically, from memory isolation, we can prove that every dereference of a field of any object in TS⋆ will only read the immediate fields of that object and will never access a prototype controlled by the adversary. This ensures that `whitelist[url]` returns true only if `url` is immediately in `whitelist`.

Undecorated TS⋆ programs can generally be given the type any (so long as they are well-scoped). Every function parameter in an unannotated TS⋆ program defaults to the type any; every `var`-bound variable is given the type of its initializer. Under this convention, in the program from §1, the type of `protect` is `any -> (any,any)-> any`, which is a subtype of any. When deploying a TS⋆ program, we assume that the JavaScript global object (the `window` object in most browsers) and all objects reachable from it are under control of the attacker. Thus, it is not safe to simply store `protect(send)` into `window.send`, since that would break memory isolation and leak a value of type `any` to un-safe code—our type system prevents the programmer from doing this by mistake.

Instead, TS⋆ provides wrappers to safely export values to the context. The TS⋆ expression `wrap(un)(protect(send))` wraps the `protect(send)` closure and the resulting term has type un, indicating that it is safe to hand to any JavaScript context while preserving memory isolation. Dually, for `e:un`, the expression `wrap(any)(`e`)` imports e safely from the context and gives it the type any.

Providing JavaScript implementations of `wrap` is non-trivial. We base our implementation on wrappers defined by Fournet et al. (2013). Their wrappers are designed to safely export statically typed values from the translation of an f⋆ program (roughly, a simply typed subset of ML) to its JavaScript context; and to import untyped values from the context into f⋆ at specific types. For example, Fournet et al.'s $\mathtt{down}_{(t*u)}$ exports a pair of translated f⋆ values (v_1, v_2) of type $(t * u)$ to the context, by building a new object with two fields initialized to $\mathtt{down}_t(v_1)$ and $\mathtt{down}_u(v_2)$. A corresponding wrapper $\mathtt{up}_{(t*u)}$ does the converse, safely copying a pair from the context and building a value that is the translation of an f⋆ pair of type $(t * u)$. Fournet et al. provide $\mathtt{up}_t$ and $\mathtt{down}_t$ wrappers for types t including `unit`, `bool`, `string`, `number`, pairs, recursive datatypes, and functions. We extend their constructions to additionally build wrappers to and from the type any.

To illustrate simple wrappers in action, we elaborate on our first example. Suppose we wished to protect `window.send` (a fictitious but simpler stand-in for JavaScript's `XMLHttpRequest` object) with an access control check. To support this, the standard library of TS⋆ provides a facility to read fields from and write fields to the global object by including the following safe interface to the `window` object implemented in JavaScript. The object `win` shadows the fields of the `window` object, safely reading and writing it within a wrapper, ensuring that the `attacker`-controlled window does not break memory isolation. Using `win` within a TS⋆ program, we can safely import `window.send`, protect it, and export it back to the context using the following snippet of code, typed in a context where the `win` object has mutable un-typed fields. Of course, the attacker may *a priori* obtain a copy, and even redefine `window.send` before our code has the chance to protect and update it, but this is an orthogonal problem, solved once for all TS⋆programs—§2.3 and §5 present our novel mechanisms to ensure first-starter privilege.

```
val win:{send: mutable un; ... }
win.send = wrap(un)(protect(wrap(any)(win.send)));
```

Wrappers are expensive, since they deeply copy the contents of objects back and forth, and—by design—they are not necessarily semantics-preserving. (For instance, they sanitize values, filter out some properties, and prevent some aliasing.) Thus, it is useful to attempt to minimize the amount of copying, while not compromising security. With this in mind, we can rewrite `protect`, adding a few types, as shown below.

```
function protect(send:(un,un) -> un) {
var whitelist={"http://www.google.com/mail":true,
"http://www.google.com/plus":true};
return function (url:string, msg:un) {
if (whitelist[url]) send(wrap(any)(url), msg);}}
win.send=wrap(un)(protect(wrap((un,un)->un)(win.send)
    ));
```

Intuitively, the `msg` argument in the closure returned by `protect` is treated abstractly, i.e., we never attempt to read from it. As such, there is no need to import that argument from the context (potentially performing a deep copy). On the other hand, the `url` argument is not abstract—it is used to project a field from the whitelist, and, as such, it had better be a string. The type system of TS⋆ gives us the flexibility to express exactly what should be imported from the context, helping us find a good balance between security and performance. The explicit use of un and wrappers `wrap`, which coerces its argument `e` to the requested type `t`, are all advances of TS⋆ relative to prior languages, e.g., f⋆, or for that matter, any prior gradually typed programming language.

2.2 Expressing invariants with assertions over runtime types

As can be expected of gradual typing, a TS⋆ program migrated from un to any can then, with some effort, be made increasingly statically typed. Static types can improve runtime safety, robustness of code, modularity, as well as provide better IDE support. Static types in TS⋆ also improve performance relative to any-typed code and, relying on RTTI, can enforce data invariants. This is enabled by *static safety* and *dynamic safety*, two properties (in addition to *memory isolation*) provided by TS⋆.

Static Safety. TS⋆ ensures that, at runtime, no failures happen during the execution of statically typed parts of the source program. Since there are no runtime checks in the compiled JavaScript for

such parts, as a bonus, the performance of statically typed TS* code will approach that of native JavaScript (and potentially exceed it, if the type information can be communicated to the VM).

Dynamic Safety. Every TS* value $v : t$ (where `t` $\neq$ `un`) is compiled to JavaScript with runtime type information (RTTI) that initially reflects v's static type t. TS* ensures that, while v's RTTI may evolve during execution (e.g., as fields are added to an extensible record), it is always (a) a subtype of v's static type t, and (b) a sound approximation (supertype) of v's most-precise type. We call this property *dynamic safety*.

As an illustration, consider the example below, which codes up a lightweight form of objects with extensible records and closures in TS*, where `point` is the type defined in §1.

```
function Point(x, y) {
var self = {};
self.x=x;
self.y=y;
self.setX=function(d:number) { self.x = d; };
return setTag(point)(self);}
```

The function `Point` creates a new point. It allocates a new empty record and stores it in the local variable `self`, then it adds three fields `x`, `y`, and `setX`. The static type of `self` is just the empty record. However, TS* allows us to add more fields to `self` than those documented in its static type. As such, the static type of a record only describes a subset of the fields in the term, as is usual with width-subtyping. Deleting fields from records is also possible—we discuss this in more detail in §3.

In the last line of `Point`, `setTag(point)(self)` checks at runtime if the content of `self` is compatible with the `point` type, and fails otherwise. The term `setTag(point)(self)` has static type `point`, although the static type of `self` remains unchanged. In order to implement checks like `setTag(point)`, every TS* term compiled to JavaScript is augmented with RTTI to record its type.

Assertions like `setTag` allow source programmers to safely update RTTI, while maintaining the runtime type invariant. Once a value has been tagged as a `point`, then it is guaranteed to always remain a point. A programmer may choose to add fields to a `point` and further update its type information (e.g., turning it into a `coloredPoint`), but it will always contain at least the fields of a `point`. Any attempt to delete, say, the `x` field, or to change it in a type-incompatible way (e.g., using a dynamically typed alias to the point) will cause a runtime error.

In contrast, statically typed code raises no such errors. TS* infers that `Point` has type `(any,any)-> point` and so the code below is statically type safe, and does not require any runtime checks.

```
var o = Point(0,0); o.setX(17);
```

As another example,[5] consider that popular web frameworks, like Dojo, provide implementations of the JSON Schema standard.[6] This allows programmers to validate JSON data, writing verbose schemas for them also as JSON objects.

For example, to describe a schema for an array of records containing pairs of user ids and names, one can write the following JSON schema.

```
schema={"type" : "object",  "properties" :
{"users" : { "type" : "array",
"items" : { "type" : "object",
"properties" : {
"id": { "type": "number" },
"user": { "type" : "string" }}}}}}
```

The JSON object `o` below matches this schema—this can be checked by calling `dojox.json.schema.validate(o,schema)`.

[5] Based on http://davidwalsh.name/json-validation.

[6] http://tools.ietf.org/html/draft-zyp-json-schema-03

In TS*, data invariants can be expressed and enforced directly using types, rather than via schemas. For example, to check that the string `"{users:[{id:1,user:"david"},{id:2,user:"walsh"}]}"` can be parsed into an array of user identities, we can write the TS* code below, assuming that `JSON.parse` has type `string -> any`. (We have programmed a similar parser in TS*; see also §6.2 for a simpler, JSON-like query parser in the Facebook API.)

```
type users = array {id:number; user:string}
function check(j:string) : users {
var o = JSON.parse(j);
if(canTag(users)(o)) { return setTag(users)(o); }
else { return []; }}
```

The schema is captured by the type `users`. We parse a string `j` as JSON using `JSON.parse`, then use the TS* operator `canTag(t)(o)` to check that `o`'s contents are consistent with `t`. If the check succeeds, we stamp `o` as a valid `users` object and return it.

2.3 Reliable primitive operations

Since the global `window` object is shared with the adversary, all objects reachable from `window` may be compromised. This includes all built-in objects provided by the VM, e.g., `Object.prototype`, `Array.prototype`, the default `String` object, and others. In order to ensure memory isolation, translated TS* programs should never read from any of those objects. This is remarkably difficult to arrange in JavaScript, since several primitive operations, e.g., reading and writing fields, depend on base prototypes, as illustrated in §1. Thus, in the face of attacker-controlled prototypes, even simple manipulations of objects are unreliable. There are two ways (that we know of) out of this conundrum.

Statically determined field accesses. In JavaScript, irrespective of `Object.prototype`, the expression `var x = {f:0}` reliably builds an object with a single field `f` initialized to `0` and assigns it to the local variable `x`. Thereafter, if we write `x.f = 1`, despite the prototype alteration, JavaScript reliably updates field `f` of `x` to `1`. These guarantees were sufficient for compiling f*, since, by typing, one can statically determine the set of fields that can be legally projected from an object. However, this approach seems infeasible for extensible records, arrays, and dynamically computed field names. Problematically, the code `var x={}; x.f=1` is not reliable, e.g. when `Object.prototype` has a setter on field `f`, and this is precisely the kind of code we would like to support (see `Point` in §2.2).

Object.defineProperty. Our alternative approach is based on the JavaScript primitive function `defineProperty`. For example, the expression `Object.defineProperty({}, "f", {value:17})` reliably adds or mutates the property `"f"` in the new object `{}` to the value `17`, without traversing the prototype chain to `Object.prototype`. Optionally, the third argument to `defineProperty` can contain metadata to make the property immutable.

We are still left with two problems. First, the VM makes `defineProperty` available as a field `window.Object`, and the adversary can simply redefine it with some other malicious function. Second, while `defineProperty` gives us a way to safely update an object, we also need a way to safely enumerate the fields of an object, e.g., to iterate over all the elements of an array.

To solve these problems, compiled TS* programs are linked with a library called `boot.js`. This library is intended to be the first piece of JavaScript that runs on a page—§5 discusses how to make a script run first, reliably, before any adversarial script. To solve the first problem, `boot.js` takes a clean copy of `defineProperty` (and many other primitives) and stores it in an immutable field. Later, the translated TS* code accesses `defineProperty` from this field, rather than `Object.defineProperty`. To address the second problem, `boot.js` defines a base object called `Q`. Later, all record and datatype values in TS* will be translated to instances of `Q`. The `Q` object provides each of its instances with a facility for safely reading, writing, and

enumerating its fields. We show a fragment of `boot.js` below, but first provide a primer on JavaScript objects.

A brief review of JavaScript's prototype-based objects. To create a new object `x` whose prototype is some other object `y`, one uses the following JavaScript recipe. Define a function, say `function Y() { this.f=17;}`; then set `Y.prototype = y`; and call `x = new Y()`. When executing `x = new Y()`, JavaScript constructs a new empty object `o` whose internal field `__proto__` is set to `y`; then calls the function `Y` with the `this` parameter bound to `o`, so that `Y` can initialize the contents of `o`. At the end, we have in `x` the object `o` with the field `f` set to `17`, unless, of course, `y` or one of its prototypes has overridden the meaning of assignment for field `"f"`.

```
1  function boot() {
2  var Q = function() { … }
3  Q.defineProperty = Object.defineProperty;
4  Q.die = function() { Q.die(); };
5  Q.prototype.set = function(p,v) {
6  Q.defineProperty(this,p,{value:v}); … return v;}
7  Q.prototype.hasField = function(f) { … };
8  Q.isTag =   function (t,t') { … }; …
9  Q.wrap =    function (t,t') { … };
10 Q.prototype.freeze(); Q.freeze();
11 Q.defineProperty(this, "Q",
12 {value:Q, writable:false, configurable:false});
13 }; boot(); boot=undefined;
```

The listing above defines a function `boot` that is run once and then discarded. Within the scope of the function, we define a constructor function `Q`. and e.g. a property `defineProperty` to keep a pristine copy of the initial `Object.defineProperty`. Line 5 defines `Q.prototype.set`, a function to reliably add a field to every `Q` object; elided elements of this function also record fields as they are added to an object, enabling us to reliably implement functions like `hasField` (line 7) that allow us to enumerate and test the existence of properties in an object. Lines 8–9 define functions that implement queries and coercions on runtime types and values. For example, source expressions like `wrap<un>(protect(send))` are translated by our compiler to a call to `Q.wrap` $(\llbracket$`(any,any)->any`$\rrbracket, \llbracket$`un`$\rrbracket)(\llbracket$`protect(send)`$\rrbracket)$, where $\llbracket t \rrbracket$ is the representation of t as a JavaScript value, defined in §3. `Q.wrap` takes the representations of both the source and the target types as parameters. Line 10 calls the JavaScript function `freeze`, to make `Q` and `Q.prototype` immutable. Finally, line 11 registers the `Q` object in `this.Q` (where `this` is the global object) and makes it immutable.

2.4 Embedding TS⋆ in JavaScript

The `Q` object in `boot.js` provides a trusted core functionality upon which we can build a secure compiler. In this section, we outline the end-to-end embedding of `protect` and `Point` within JavaScript. There are a few broad features of the translation that we focus on:

- Translating all objects to instances of `Q`.
- Adding runtime-type information to every object and function.
- Checking runtime type information in the `any` fragment.
- Embedding wrappers to safely export/import values.

The listing below shows the translation of the TS⋆ function `Point` to JavaScript. The translated code is placed within a single enclosing function to introduce a fresh local scope. Without this, TS⋆ definitions would implicitly leak into the global un-typed object. The type annotations in TS⋆ are all erased in JavaScript.

```
1  function () {
2  var Point = function (x,y) {
3  var self = new Q();
4  self.set("rtti", ⟦{}⟧);
5  write(self, "x", x);
6  write(self, "y", y);
7  var tmp=function(d:number) {write(self, "x", d);}
```

Value	v	::=	$x \mid$ `true` $\mid$ `false` $\mid \lambda x{:}t.e \mid D\ \bar{v}$
Expr.	e	::=	$v \mid \{\bar{f} = \bar{e}\} \mid e.f \mid e.f = e' \mid e[e'] \mid e[e'] = e''$
		$\mid$	**let** x = e **in** $e' \mid e\ e' \mid D\ \bar{e}$
		$\mid$	**if** e **then** e' **else** $e'' \mid \langle q\ t\rangle e \mid \langle c\ t\rangle e$
Query	q	::=	*isTag* $\mid$ *canTag* $\mid$ *canWrap*
Coercion	c	::=	*setTag* $\mid$ *wrap*
Type	t,u	::=	`bool` $\mid$ `T` $\mid$ `any` $\mid$ `un` $\mid t \to u \mid \{\bar{f} :^{\bar{a}} \bar{t}\}$
Access	a	::=	`r` $\mid$ `w`
Sig.	S	::=	$. \mid D{:}\bar{t} \to T \mid S, S'$
Env.	Γ	::=	$. \mid x{:}t \mid \Gamma, \Gamma'$

Figure 1. Formal syntax of TS⋆

```
8  Q.defineProperty(tmp, "rtti", ⟦number → unit⟧);
9  write(self,"setX", tmp);
10 return Q.setTag(⟦any⟧, ⟦point⟧)(self); }
11 Q.defineProperty(Point, "rtti", ⟦(any,any) → point⟧);
12 var o = Point(0,0);
13 o.setX(17);
14 }()
```

Line 3, the source empty record `{}` is compiled to a new `Q` object. Line 4, we set the `rtti` field of `self` to the translation of a source empty record. Lines 5 and 6, we use the macro *write* to add two properties to the `self` object. This macro (defined in §3) checks that the RTTI of the assigned field (if any) is compatible with RTTI of the assignee. In this case, since the `rtti` field of `self` is just the empty record, it does not constrain the contents of any of its fields—so, these assignments succeed, and the fields are added to `self`. Line 7, we translate `setX`, and at line 8, we tag it an `rtti` field recording its source type. We then add it to `self` using *write*. Line 10, the call to `Q.setTag` checks whether `self` contains a valid representation of a source `point`. For this, it examines the representation of the type $\llbracket$`point`$\rrbracket$; notices that the type requires three fields, `x`, `y`, and `setX`; then checks if the `self` object contains values in those three fields whose RTTI is compatible with the request types, `number`, `number`, and `number -> unit`, respetively. Once this check succeeds, `setTag` updates the `rtti` field of `self` to $\llbracket$`point`$\rrbracket$. An invariant of our translation is that the `rtti` field of every object evolves monotonically with respect to the subtyping relation. That is, `self.rtti` was initially $\llbracket\{\}\rrbracket$ and evolves to $\llbracket$`point`$\rrbracket$, where `point <: {}`. The RTTI of `self` may evolve further, but it is guaranteed to always remain a subtype of `point`. Line 11, we add an `rtti` field to the `Point`. Finally, at lines 12 and 13 we see the translation of a statically typed fragment of TS⋆. Pleasantly, the translation there is just the identity.

As shown, the translated program does not interact with its context at all. However, the programmer can choose to export certain values to the context. For example, including the top-level declaration `val Point:(any,any)-> point` in the source program instructs the compiler to wrap and export `Point` to the context by inserting the following code after Line 13.

```
win.Point = Q.wrap(⟦(any,any) → point⟧, ⟦un⟧)(Point);
```

3. Formalizing TS⋆

This section formalizes TS⋆ by presenting its type system and type-directed translation to JavaScript. We describe in detail our runtime representation of types and the JavaScript implementations of `Q.wrap`, `Q.setTag` and related functions that manipulate translated terms and their types. We conclude this section with a detailed comparison of TS⋆ with prior gradual type systems.

3.1 Syntax

Figure 1 presents our source syntax. To aid in the readability of the formalization, we employ compact, λ-calculus style notation, writing for example $\lambda x{:}t.e$ instead of `function (x:t){return e;}`.

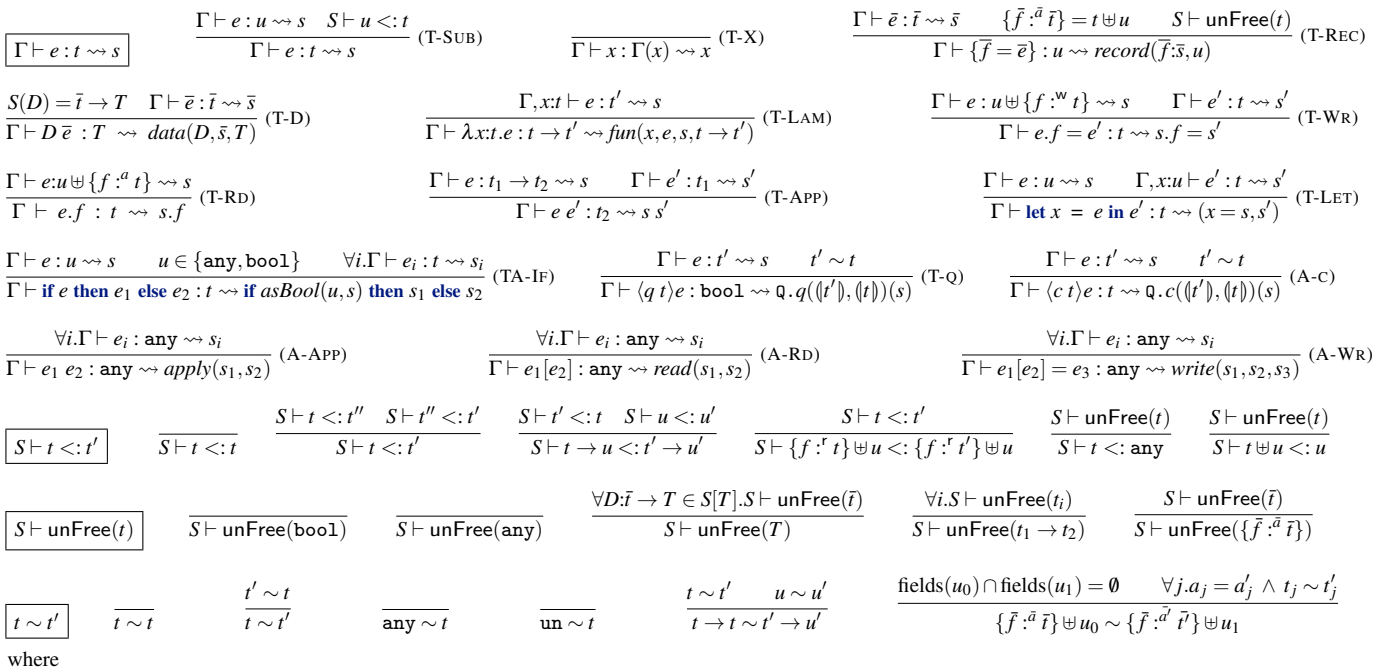


Figure 2. A type-directed translation of TS* to JavaScript

We also write $\overline{e}$ for a sequence of expressions $e_1, \ldots, e_n$, $f(\overline{e})$ for the application $f(e_1, \ldots, e_n)$, and so on.

Values v include variables x, booleans, typed λ-abstractions, and data constructors D applied to a sequence of values. For conciseness, we exclude primitives like numbers and strings, since they can in principle be encoded using data constructors. Of course, in practice, our implementation supports JavaScript primitives, and so we use them in our examples.

In addition to values, expressions e include record literals, projections of static fields, and assignments to static fields. We also include projections of computed fields $e[e']$ and assignment to computed fields $e[e'] = e''$. It is important to note that records, even records of values, are not values; as in JavaScript evaluating a record returns the heap location where the record value is stored. We have `let`-bindings (corresponding to immutable `var` bindings in our concrete syntax); function application; data constructor application; and conditionals. Finally, we have RTTI-based query operations $\langle q\ t \rangle e$, and coercions $\langle c\ t \rangle e$.

Types t, u include a number of primitive types (`bool` for boolean values, and `any` and `un` for dynamic values), abstract data types ranged over by T, and records. Record types are written using the shorthand $\{\overline{f} :^{\overline{a}} \overline{t}\}$ to denote the type $\{f_1 :^{a_1} t_1, \ldots, f_n :^{a_n} t_n\}$ where the f_i are distinct and the a_i are accessibility annotations: `r` for read-only, and `w` for mutable. We also write $t \uplus u$ for the record type $\{\overline{f_1} :^{\overline{a_1}} \overline{t_1}, \overline{f_2} :^{\overline{a_2}} \overline{t_2}\}$, where $t = \{\overline{f_1} :^{\overline{a_1}} \overline{t_1}\}$ and $u = \{\overline{f_2} :^{\overline{a_2}} \overline{t_2}\}$.

The type system is given with respect to a signature S which maps data constructors D to their type signature, written $\overline{t} \to T$. In places we need to refer to *all* the data constructors for a given abstract data type T in the signature S. We use the shorthand $S[T]$ which is defined as $\{D : \overline{t} \to T \mid S(D) = \overline{t} \to T\}$. We also have a standard type environment Γ binding variables to their types.

Although we have data constructors, pattern matching in TS* is not primitive. Instead, it can be encoded in terms of the other constructs, as defined below. Note, we freely use `&&` and the other boolean operators, as well as `==`, structural equality on TS* values.

$\mathbf{match}\ e\ \mathbf{with}\ D_{\overline{t} \to T}\ \overline{x}\ \to e_1\ \mathbf{else}\ e_2 \triangleq$
$\mathbf{let}\ y = e\ \mathbf{in\ if}\ (\langle isTag\ T \rangle y\ \mathtt{\&\&}\ y.c == \mathtt{"D"})\ \mathbf{then\ let}\ \overline{x} = \overline{y[\mathtt{i}]}\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2$

3.2 Type system and translation

Figure 2 defines the judgment $\Gamma \vdash e : t \rightsquigarrow s$, which states that in an environment Γ (along with an implicit signature S), TS* expression e can be given the type t and translated to the JavaScript program s. We present the type system declaratively—making it algorithmic in our implementation, given reasonable default type annotations, is not hard. We plan to add more systematic bidirectional type inference in the near future.

At a high level, the type system is designed to enforce the following three properties mentioned in §2:

Static safety. TS* programs have no failing dynamic checks during the execution of statically typed sub-terms. We achieve this via two

mechanisms: (a) the rules prefixed by (T-) enforce the static typing discipline and they never insert any runtime checks when compiling the program; (b) when a value v:any is passed to a context that expects a precise type, e.g. `point`, the compiler inserts instrumentation to ensure that v is indeed at least a `point`. Instrumentation inserted elsewhere in dynamic code also ensures that v henceforth remains at least `point`. This protects the statically typed code from future modifications to v. In the other direction, the type system allows for v :`point` to be passed to a `any`-typed context via subtyping.

Dynamic safety. The RTTI of v:t is always a subtype of t and a sound approximation of v's most precise type—by two mechanisms: (a) v's RTTI initially reflects t and the *setTag* operation ensures that RTTI always evolves towards the more precise types per subtyping, and (b) the rules prefixed by (A-) instrument the translation of the `any`-typed parts of the source to enforce that modifications to v respect its RTTI. (We envision that an IDE can highlight uses of (A-) rules to the programmer as potential failure points.)

Memory isolation. `un`-typed code cannot directly access an object reference that TS* code may dereference, enforced by ensuring that programs treat the `un` type abstractly. The only way to manipulate `un` values is via defensive wrappers, which means that typed code never dereferences an `un`-typed memory location, and that `any`-typed references can never be handed directly to the adversary. The subtyping rules are designed to prevent masking the presence of `un`-values in records using width-subtyping or subtyping to `any`.

We now turn to describing each of the rules in detail. The first rule in the judgment, (T-Sub), is a subsumption form which shows that a use of subtyping in TS* does not change the translation of a term. The subtyping relation $S \vdash t <: t'$ (also in Figure 2) is mostly standard. Depth subtyping on records is permitted only for immutable fields. The last but one rule allows all types that do not contain the `un` type to be a subtype of `any`. (The auxiliary predicate `unFree` detects occurrences of `un` in a type.) Allowing `un <: any` would clearly break our invariants. Allowing `{f:un} <: any` is also problematic, since if a value `v:{f:un}` could be promoted to `v:any`, then `v["f"]` would also have type `any`, even though it produces an untrusted value. The last subtyping rule provides width-subtyping on records, forgetting the fields to weaken $t \uplus u$ to u, only so long as t contains no occurrences of `un`.

The rule (T-X) for typing variables is standard. (T-Rec) introduces a record at type u, such that u includes all the `un`-fields (necessary for compatibility with subtyping). Its compilation allocates a new `Q` object, safely sets the fields $\bar{f}$ to $\bar{s}$, and finally adds an `rtti` field containing $(\!|u|\!)$. The rule (T-D) for typing data constructors is similar. The typing of functions with (T-Lam) is standard; however, the translation to JavaScript is a bit subtle: it defines a JavaScript function tagged with an `rtti` field, whose body s is preceded by declarations of all the `let`-bound variables in e, the source function body. These (and other) rules use the JavaScript form $(\overline{e})$, which evaluates every e_i in $\overline{e}$ and returns the last one.

The rules (T-Wr), (T-Rd), (T-App), (TA-If), and (T-Let) are standard. One slight wrinkle in (TA-If) is that in case the guard has static type `any`, we insert a check to ensure that it is a boolean at runtime—we could have split the rule into two.

The rules (T-Q) and (A-Q) cover the RTTI-based query $\langle q\ t \rangle e$, and coercion $\langle c\ t \rangle e$. In each case, we have an expression $e : t'$ compiled to s, and we apply q or c at type t, so long as t and t' are compatible. Type compatibility is a simple reflexive, symmetric, and *non*-transitive relation, similar to Siek and Taha (2006).

Reflecting on RTTI The form $\langle isTag\ t \rangle e$ is compiled to `Q.isTag(`$(\!|t'|\!)$`,`$(\!|t|\!)$`)(s)`, implemented in `boot.js`. It checks whether the RTTI of s exists and if it is a subtype of t, returning a boolean. `Q.canTag(`$(\!|t'|\!)$`,`$(\!|t|\!)$`)(s)` checks whether the least runtime type of `s` is a subtype of t. If either t or t' is `un`, `canTag` returns false. For records and datatypes, `canTag` recursively examines the contents of `s`, and checks the RTTI of functions, which must be present on every non-`un` function. If `Q.canTag(`$(\!|t'|\!)$`,`$(\!|t|\!)$`)(s)` succeeds, `Q.setTag(`$(\!|t'|\!)$`,`$(\!|t|\!)$`)(s)` also succeeds and returns s after mutating its RTTI (recursively) to be at least t. On a failure, `Q.setTag(`$(\!|t'|\!)$`,`$(\!|t|\!)$`)(s)` terminates the execution.

Securely importing from `un` The operator $\langle wrap\ \texttt{any} \rangle e$ imports e:`un`, compiling to `Q.wrap(`$(\!|\texttt{un}|\!)$`,`$(\!|\texttt{any}|\!)$`)(s)`. It begins by determining s's simple type using JavaScript's `typeof`. If `s` is a string, it uses the Fournet et al.'s `upstring`; other primitive types are similar. If `s` is an object, `wrap` creates a new `Q` object `x`, enumerates all the fields `f` of `s` and adds them to `x` after recursively wrapping the field value to `any`. If `s` is a function, `wrap` applies Fournet et al.'s `upfun` to safely import a function at the type `any -> any`. To a first approximation, this places a higher-order wrapper around `s` of the following form: `function (x){ return Q.wrap(`$(\!|\texttt{un}|\!)$`,`$(\!|\texttt{any}|\!)$`)(s(Q.wrap(`$(\!|\texttt{any}|\!)$`,`$(\!|\texttt{un}|\!)$`)(x))); }`. In reality, `upfun` is more complex, since it ensures memory isolation in the presence of stack walks using JavaScript's `argument.callee.caller`. The general case, when $t \neq \texttt{any}$, is similar.

Securely exporting to `un` The implementation of `Q.wrap(`$(\!|t|\!)$`,`$(\!|\texttt{un}|\!)$`)(s)` for $t \neq \texttt{un}$, exports `s` to the context. Since `s` is a TS* value, `wrap` inspects its RTTI. If s is a primitive, it uses one of Fournet et al.'s wrappers, e.g., `downstring`. If `s` is an object, `wrap` creates an empty object literal `x`, enumerates all the fields `f` of `s`, exports them recursively and adds them to `x`. For functions, it uses Fournet et. al.'s `downfun` to export it at type `Un -> Un`.

TS* allows wrapping between non-`un` types as well, e.g. from type `t -> u` to type `t' -> u'`. Such wrappers can be used when RTTI operations seem too strict. For example, `setTag` on a function f with RTTI `any -> any` at type `bool -> bool` would fail, since `any -> any` $\not<:$ `bool -> bool`. However, a wrapper from `any -> any` to `bool -> bool` succeeds by placing a higher-order wrapper around f.

Instrumenting `any`***-typed code*** The remaining (A-) rules instrument the translated programs to ensure safety. In (A-App), we first check that s is a function. Then, before calling the function, we tag the argument with the type of the function's parameter. (A-Rd) simply checks that s_1 is an object and that s_1 has field s_2. (A-Wr) checks that s_1 is an object. It then checks that s_1's RTTI allows for field s_2 to be written. If s_1's RTTI does not contain s_2, it is treated as a new property addition—deleting a property if it is not present in the RTTI is also straightforward, although we do not cover it here. Otherwise, it should contain a mutable s_2 field, and before writing, s_3 is tagged with the type expected by s_1's RTTI. In all cases, failed runtime checks call `Q.die`, which exhausts the JavaScript stack (since it does not optimize tail calls). This is drastic but effective; friendlier failure modes are feasible too.

3.3 Discussion and related work on gradual typing

Languages that mix static and dynamic types go back at least to Abadi et al. (1991) and Bracha and Griswold (1993). Gradual typing is a technique first proposed by Siek and Taha (2006), initially for a functional language with references, and subsequently for languages with various other features including objects. Several others have worked in this space. For example, Flanagan (2006) introduces hybrid typing, mixing static, dynamic and refinement types; Wadler and Findler (2009) add blame tracking to a gradual type system; Herman et al. (2010) present gradual typing with space-efficient wrappers; Bierman et al. (2010) describe type dynamic in C$^\sharp$; and Ina and Igarashi (2011) add gradual typing to generic Java.

Our system is distinct from all others in that it is the first to consider gradual typing for a language *embedded* within a larger, potentially adversarial environment via the type `un`. We are also, to the best of our knowledge, the first to consider gradual types as a means of achieving security.

To compare more closely with other systems, set aside un for the moment, and focus on the interaction between any and statically typed TS⋆. Previous type systems mediate the interactions between static- and any-typed code by *implicitly* attaching casts to values. Higher order casts may fail at a program point far from the point where it was inserted. To account for such failures, blame calculi identify the cast (with a label to indicate the term or context) that causes the failure—Siek et al. (2009) survey blame calculi based on the errors they detect, points of failures, and casts they blame

Interactions between static- and any-typed TS⋆, in contrast, is based primarily on RTTI. Casts (wrappers in our terminology) are never inserted implicitly, although they may be inserted explicitly by the compiler. This design has the following advantages.

Preservation of object identity. Object identity in JavaScript is a commonly used feature. Since TS⋆ does not *implicitly* attach casts to values, it never implicitly breaks object identity in the source during compilation. Previous gradual type systems, with implicit casts, would always break object identity.

Space efficiency. Casts can pile up around a value, making the program inefficient. Herman et al. (2010) introduce a novel cast-reduction semantics to gain space-efficiency. TS⋆'s RTTI is also space efficient (there's only one RTTI per object), but does not require cast-reduction machinery.

Runtime reflection. JavsScript provides a coarse reflection mechanism using its `typeof` operator. RTTI tags in TS⋆ enrich runtime reflection by allowing programmers to inspect types more precisely.

Static safety and eager failures. In contrast to our RTTI-based mechanism, statically typed code in other gradual type systems could fail (although blame would help them ascribe it to the any-typed code). Consider the following TS⋆ example.

```
(λr:{f:ᵂint}. r.f) {f:true}
```

Compiling this term using (A-APP) introduces a `setTag` on the argument at type `{f:ᵂint}`. The `setTag` operation, at runtime, recursively checks that `v` is indeed `{f:ᵂint}`, and expectedly fails. Thus, the failure happens prior to the application, a failure strategy called *eager* in prior works. Herman et al. (2010) also argue that their system can provide eager failures, but transposed to their notation (with the record replaced by a `ref any`), the failure occurs at the property read within the statically typed λ-term, breaking static safety. When eager runtime checking seems too strict, TS⋆ wrappers provide an escape hatch. Arguably, for our security applications, a predictably uniform eager-failure strategy is a suitable default.

In their calculus the example would be:

```
let x:Ref int -> int = (λr:Ref int. !r) in
let v:any = true in let l:Ref any = ref v in
x l
```

In their calculus, the application `x l` just inserts a coercion from `Ref any` to `Ref int`, that succeeds, even in the eager semantics. The failure happens inside the λ, which is a statically typed fragtment, when `r` is dereferenced. Thus, violation of static safety property.

We plan to pursue providing eager semantics even for wrappers as future work.

For wrapping between non-un types, we can use refinements in RTTI to provide more precise type checking. For wrapping from un types, we can provide a *safe* `eval`, compiled with TS⋆, that imports un-typed values with eager failure semantics.

Dynamic safety and blame. With no failures inside statically typed code to explain at runtime, our RTTI-based scheme may seem to obviate the need for blame. However, because we enforce dynamic safety (RTTI evolves monotonically), failures may now arise in any-typed code, as in the following example.

```
let v:any={f:2} in (λr:{f:ᵂint}. r.f) v; v.f = "hi"
```

This time, the `setTag` of `v` to `{f:ᵂint}` succeeds, and it modifies `v`'s RTTI to be `{f:ᵂint}`. But now, the update of `v.f` to `"hi"` fails. This failure in the any-typed fragment should be *blamed* on the `setTag` operation instrumented at the application. We plan to pursue the details of this seemingly new notion of blame as future work.

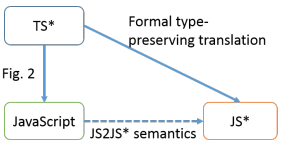
4. Metatheory of the translation

This section formally establishes memory isolation, static safety, and dynamic safety for TS⋆ programs translated to JavaScript. Clearly, such a proof requires a formal semantics for JavaScript—we rely on JS⋆, a translation semantics for JavaScript developed by Swamy et al. (2013), which is in turn based on λJS (Guha et al. 2010). We provide a brief review of JS⋆, define the central invariants of our translation, and describe our main theorem. A full development of the formalism, including all proofs, is available in the supplementary material.

4.1 A review of JS⋆ and the high-level proof strategy

JS⋆ is a subset of F⋆ (Swamy et al. 2011), a dependently programming language whose type system allows expressing functional-correctness properties of higher-order programs, whose effects may include non-termination, higher-order state, exceptions, and fatal errors. These features enable it to describe the features of JavaScript, and Swamy et al. provide a tool called JS2JS⋆ to translate JavaScript concrete syntax to JS⋆. The JS⋆ semantics for JavaScript has been used previously both as a means of verifying JavaScript source programs (after translation to JS⋆) as well as in Fournet et al.'s proof of full abstraction from f⋆ to JavaScript. At its core, JS⋆ provides a mechanically verified library called *JSVerify* that tries to faithfully model most security-relevant details of JavaScript, including, for example, its object model and calling convention. The metatheory of TS⋆ is stated in terms of its translation to JS⋆, i.e., programs that can be type-checked against the *JSVerify* API. The validity of our theorem depends on JS⋆ being a faithful model of JavaScript, an assumption that can be checked separately, e.g., by semantics testing.

To set up the formal machinery, we develop a model of our compiler by transposing the translation judgment in Figure 2 to instead generate JS⋆ code (see supplementary material). The relationship among these translations is depicted alongside. The translation from TS⋆ to JS⋆ can be seen as the composition of the translation from TS⋆ to JavaScript, and then from JavaScript to JS⋆. Our main theorem is stated as a type-preservation result from TS⋆ to JS⋆, where the types in JS⋆ are precise enough to capture our desired invariants, i.e., static safety, dynamic safety, and memory isolation.



Monadic computation types with heap invariants. All computations in js⋆ are typed in a state monad of predicate transformers, *iDST*, which is parameterized by a heap-invariant predicate *HeapInv* and a heap-evolution predicate δ. The type *iDST a WP* is the type of a computation, which for any post-condition *post*, when run in an initial heap *h*, may diverge or else produce a result *v*:*a* and a final heap *h*′ that satisfy the formula *post v h*′∧ *HeapInv h*′ ∧ δ *h h*′, so long as *HeapInv h* ∧ *WP post h* is valid. Additionally, all intermediate heaps in the computaion satisfy *HeapInv*, and every intermediate heap is related to all its successors by δ. That is, in *iDST a WP*, *a* is the result type, and *WP* is a predicate transformer that computes a pre-condition for the computation with respect to *post*, any desired post-condition; *HeapInv* is an invariant on a heap, and δ is a reflexive and transitive relation constraining how the heap evolves. Formally, the types and our theorem also account for exceptions and fatal errors; however, we gloss over them here for lack of space.

The main idea behind our proof is that a TS* term $e{:}t$ is translated to a JS* term e' of type $iDST\ dyn\ WP_{[\![t]\!]}$, where

$$WP_{[\![t]\!]} = \Lambda post.\lambda h.\ Ok\ (loc(e))\ h \wedge \forall v\ h'.\ [\![t]\!]\ v\ h' \Longrightarrow post\ r\ h'$$

This ensures that if e' is run in an initial heap h satisfying $HeapInv\ h$ and $Ok\ (loc(e))\ h$ (meaning that all free-variables of the source-term e are correctly promoted to the heap in JS*); then, it will either (1) terminate with a result v and a final heap h' satisfying $[\![t]\!]\ v\ h'$; or (2) it will diverge or raise an exception. All the while during the execution of e', $HeapInv$ will be true, and the heap will evolve according to δ. This result holds even when e' is linked with arbitrary JavaScript code—adapting the universal typability lemma of Fournet et al. (2013), JavaScript code can always be typed in JS* at a type corresponding to `un`.

Our main task then is to carefully define $HeapInv$, δ and $[\![t]\!]$ such that they capture our desired invariants, and then to prove that translated programs are well-typed in JS*.

4.2 Invariants of the translation

To prove memory isolation, JS* provides a partitioned heap model. Every object reference $l{:}loc$ carries a tag, $l.tag$, which records the name of the compartment into which the reference points, i.e., each compartment is a disjoint fragment of the domain of the heap. There are six compartments in the heap model. The *Ref* compartment holds objects corresponding to TS* records, datatypes, and RTTI; the *Abs* compartment holds function objects; the Q compartment holds the base prototype object initialized and then frozen by `boot.js`; and the *Un* compartment belongs to the adversary. We focus primarily on properties of these first four compartments. The remaining two compartments, *Inv* and *Stub*, are inherited unchanged from Fournet et al. (2013)—the former is for maintaining local variables, and the latter for tracking function objects used to make safe callbacks to the attacker.

Refined type dynamic. All JavaScript values (including the translation of TS*) are represented as JS* values of type *dyn*, defined below. We show only three representative cases. $d{=}Str\ s$ is an injection of $s{:}string$ into type *dyn*, where the refinement $TypeOf\ d{=}string$ recalls the static type. For object references $Obj\ (l{:}loc)$, the refinement is l's tag, i.e., $\{Ref, Abs, Un, Q\}$. Finally, for functions, $Fun\ o\ f$ builds a value of type *dyn* from a function closure f and the JavaScript object o for that closure. Its refinement is the predicate transformer of f.

```
type dyn = ...
| Str: string →d:dyn{TypeOf d=string}
| Obj: l:loc →d:dyn{TypeOf d=l.tag}
| Fun: ∀WP. o:dyn →(this:dyn →args:dyn →iDST dyn (WP o args this))
→d:dyn{TypeOf d=WP}
```

Translation of types. To recover the precision of TS* types in JS*, we translate source types to predicates on *dyn*-typed values and the heap: $[\![t]\!]\ d\ h$ states that the value $d{:}dyn$ is the translation of a source value of type t in the heap h. The translation of a type is with respect to a heap, since a source value allocated at the type $\{f :^a number\}$, may evolve to become a value of type $\{f :^a number, g :^a number\}$ in some subsequent heap. This is in contrast to, and a significant generalization of, the translation of F* to JS*, where a value's type does not evolve and is not suject to subtyping.

$$
\begin{array}{lcl}
[\![\texttt{string}]\!]\ d\ h & = & TypeOf\ d{=}string \\
[\![\texttt{un}]\!]\ d\ h & = & IsUn\ d \\
[\![t]\!]\ d\ h & = & \exists u <: t.\ Tagged\ u\ d\ h \quad \text{if } t \notin \{\texttt{string, un}\} \\
Tagged\ u\ d\ h & = & \texttt{"rtti"} \in dom\ h[d] \wedge Rep\ u\ (h[d][\texttt{"rtti"}])\ h \wedge Is\ u\ d\ h \\
Is\ \texttt{any}\ d\ h & = & Prim\ d \vee Fun\_is\ d \vee (TypeOf\ d{=}Ref \wedge restAny\ \{\}\ d\ h) \\
Is\ T\ d\ h & = & TypeOf\ d{=}Ref \wedge \bigvee_{D_{\bar{t}\to T}} h[d][\texttt{"c"}]{=}Str\ \texttt{"D"} \\
 & \wedge & \wedge_i\ [\![t_i]\!]\ (h[d][i])\ h \wedge restAny\ \{1..n\}\ d\ h \\
Is\ \{\bar{f} :^{\bar{a}} \bar{t}\}\ d\ h & = & TypeOf\ d{=}Ref \wedge \bar{f} \subseteq dom\ h[d] \\
 & \wedge & [\![\bar{t}]\!]\ h[d][\bar{f}]\ h \wedge restAny\ \bar{f}\ d\ h \\
Is\ (t_1 \to t_2)\ d\ h & = & TypeOf\ d = \lambda o\ args\ this.\Lambda p.\lambda h. \\
 & & \quad [\![t_1]\!]\ h[args][\texttt{"0"}]\ h \wedge \forall r\ h'.\ [\![t_2]\!]\ r\ h' \Longrightarrow p\ r\ h' \\
restAny\ fs\ d\ h & = & \forall f{\in}dom(h[d])\backslash fs.\ \neg Reserved\ f \Longrightarrow [\![\texttt{any}]\!]\ h[d][f]\ h
\end{array}
$$

Since strings are immutable $[\![\texttt{string}]\!]\ d\ h$ does not depend on h. Likewise, an `un`-typed value always remains `un`-typed—we define $IsUn$ shortly. For other types, $[\![t]\!]\ d\ h$ captures the subtyping relation, stating that there exists a type $u <: t$ such that the value's `rtti` is tagged with the runtime representation of u (the predicate $Rep\ u\ (h[d][\texttt{"rtti"}])\ h$), and $Is\ u\ d\ h$, i.e., the value d can be typed at u in h. $Is\ \texttt{any}\ d\ h$ states that d is either a primitive (e.g., a string), a function, or a location in the *Ref* heap where all its non-reserved fields (excluding, e.g, `"rtti"`) are typeable at `any` ($restAny$). For datatypes and records, we require d to be a location in the *Ref* heap, with the fields typed as expected, and with all other fields not mentioned in the type being `any`. The case for functions is most interesting. $Is\ (t1 \to t2)\ d\ h$ states that the d's predicate transformer builds a pre-condition that requires the first argument to satisfy $[\![t_1]\!]$ (all JavaScript functions are variable arity, receiving their arguments in an array; however a TS* function will only read the first one). In return, the predicate transformer ensures that the result r (if any) will satisfy $[\![t_2]\!]$. As mentioned earlier, we have elided our formal treatment of exceptions.

***Un* values.** The predicate $IsUn\ v$ defines when the value v could have been produced by, can be given to, or is accessible by the context. *Un* values include primitives; references to objects in the *Un* heap; or the immutable Q object (which is reachable from the global object). Additionaly, *Un* values can be functions whose specification indicates that it takes *Un* arguments to *Un* results.

$$IsUn\ x \triangleq TypeOf\ x \in \{bool, string, float, Un, Q\} \vee TypeOf\ x{=}Un2Un$$
$$Un2Un \triangleq \lambda o\ args\ this\ post\ h.\ IsUn\ o \wedge IsUn\ this \wedge IsUn\ args$$
$$\wedge\ (\forall\ r\ h'.\ IsUn\ r \Longrightarrow post\ r\ h')$$

***HeapInv*, the global heap invariant.** Our main heap invariant is a property of every location x in the heap. Its full definition contains seven clauses; we show the most important ones.

$$
\begin{array}{lcl}
HeapInv\ h & \triangleq & \forall x.\ x \in dom\ h \Longrightarrow \\
(1) & & (x.tag{=}Un \Longrightarrow \forall x \in dom\ h[x].\ IsUn\ h[x]) \\
(2) & \wedge & (x.tag{\in}\{Ref,Abs\} \wedge \texttt{"rtti"} \in dom\ h[x] \Longrightarrow \exists t.\ Tagged\ t\ d\ h) \\
(3) & \wedge & (x.tag{=}Ref \Longrightarrow IsQ\ h[x]) \\
(4) & \wedge & (x.tag{=}Q \Longrightarrow QSpec\ h[x]\ h)
\end{array}
$$

$$
\begin{array}{lcl}
IsQ\ o\ h & \triangleq & TypeOf\ o[\texttt{"@proto"}]{=}Q \wedge \texttt{"fields"} \in dom\ o \\
 & \wedge & \forall f \in AsSet\ o[\texttt{"fields"}]\ h.\ f \in dom\ o \\
 & \wedge & \forall f \in dom\ o.\ Reserved\ f \vee f \in AsSet\ (o[\texttt{"fields"}])\ h
\end{array}
$$

Clause (1) asserts that all the contents of an *Un* object are also *Un*. Clause (2) asserts that every object in the *Ref* and *Abs* compartment with an `"rtti"` field is tagged properly. Clause (3) additionally specifies that every object in the *Ref* heap is an instance of Q. The predicate $IsQ\ o\ h$ on an object establishes that o's prototype points to the heap compartment, Q; and that it contains a `"fields"` property that accurately models all the fields in the object—useful for testing the existence of and enumerating properties. Clause (4) asserts that the object in the Q compartment has a specification described by $QSpec$, which gives a type to each of its fields.

Within this invariant are two key properties of TS⋆. The first clause guarantees that the only values reachable from a location in the *Un* heap are themselves `un`-values. Therein lies our memory isolation property—the adversary can never meddle with TS⋆ objects directly, since these reside in the *Ref* and *Abs* heap, which are disjoint from *Un*. The invariant, in its second clause, also captures dynamic safety, i.e., every object in the *Ref* and *Abs* heap, once tagged with RTTI are properly typed according to it.

δ, the heap evolution invariant. The full definition of δ has 4 clauses; we show the main one below: δ ensures that, for all objects in the *Ref* and *Abs* heaps, their `"rtti"` fields only evolve "downward" in the subtyping hierarchy.

$\delta\ h0\ h1 \triangleq \forall l \in dom\ h0, t_0, t_1.$
$l.tag \in \{Ref,Abs\} \wedge Rep\ t_0\ h0[l][\texttt{"rtti"}]\ h0 \wedge Rep\ t_1\ h1[l][\texttt{"rtti"}]\ h1$
$\Longrightarrow t_1 <: t_0$

A relatively easy lemma derivable from these definitions implies our static safety property. In particular, Lemma 9 guarantees that if a value v (potentially a location to a heap-resident record) is typeable at type t in some initial heap h_0 then as the program's heap evolves according to δ, v remains typeable at t. This ensures that it is safe for TS⋆ to ascribe a value a static type, since that type is an invariant of the value at runtime.

Lemma 1 (Static safety: δ preserves the interpretation of types)**.** *For all values v, heaps h_0 and h_1 such that HeapInv h_0, HeapInv h_1 and $\delta\ h_0\ h_1$, if for some t we have $[\![t]\!]\ v\ h_0$ then $[\![t]\!]\ v\ h_1$*

Finally, our main theorem, as promised, is a type preservation result that guarantees memory isolation, dynamic safety and static safety for TS⋆ programs translated to JS⋆. The relation $\Gamma \vdash_f e : t \rightsquigarrow e'$ is the formal translation of TS⋆ to JS⋆ (the index f is the name of the current function object in e'; a technical detail).

Theorem 1 (Type preservation)**.**
Given a TS⋆ context Γ, expression e and type t,
if $\Gamma \vdash_f e : t \rightsquigarrow e'$ for some JS⋆ expression e' and function object f,
then $[\![\Gamma]\!], f{:}dyn, \Gamma_{loc(e)} \vdash e' : iDST\ dyn\ WP_{[\![t]\!]}$

We conclude our formal development with a few remarks on the scope of our theorem and the style of its proof. First, our result is applicable to the translation of TS⋆ to JavaScript only inasmuch that JS⋆ is an accurate model of all of JavaScript—Fournet et al. (2013) argue for how JS⋆ is an adequate model of all security-relevant features of JavaScript. Regardless, the availability of these semantics together with its program logic is what made our proof feasible. Given an operational semantics, but lacking a program logic, our proof would have been mired in tedious inductions over the operational semantics. With JS⋆, we were able to carry out our proof as an induction over the compilation relation, and use the type system of JS⋆ to structure and prove our invariants, which is what renders the result tractable.

Second, our result includes within its trusted computing base the correspondence of `boot.js` to the *QSpec* specification in the last clause of *HeapInv*. While it is perhaps standard for compiler and verification projects to rely on a small amount of trusted code, we would like to do better. In particular, we aim to use the JavaScript verification toolchain developed by Swamy et al. (2013) to verify `boot.js` for compliance with *QSpec*—at the time of writing, this was still incomplete. More substantially, we would also like to build a translation validation pipeline for our compiler implementation, reflecting the generated JavaScript back into JS⋆ for verification, i.e., we would like for our compiler implementation to also be formally type-preserving.

5. The TS⋆ compiler and its safe deployment

We have built a prototype compiler from TS⋆ to JavaScript and used it to compile all our examples. The compiler consists of several phases, reusing some infrastructure provided by the F⋆ compiler. It first translates source programs in concrete TS⋆ syntax to F⋆, using a variant of the DJS parser (Bhargavan et al. 2013). The second phase is a type-directed F⋆ to JavaScript translation (an algorithmic adaptation of Figure 2), which reuses parts of the JavaScript backend of F⋆. We wrote afresh or modified 800 line of OCaml and 1,800 lines of F#.

We evaluated our compiler by gradually migrating JavaScript sources to TS⋆, while ensuring that the migrated code (after compilation) exports the same API as the original. The compiled JavaScript code for our examples is about 4X as big as the original TS⋆ programs—we can reduce most of the code size overhead by removing the A-normal form our compiler emits. We have yet to conduct a thorough performance analysis of our compiler, nor have we implemented any optimizations. But, as mentioned previously, statically typed TS⋆ should incur little, if any, runtime overhead. Understanding and optimizing the performance profile of `any`-typed code is left as future work.

The table below lists our examples and compares the code size of the generated JavaScript programs with those of the source TS⋆ programs. We have discussed all examples previously, except for Csrf-whitelist which implements CSRFGuard using a white list of URLs. Columns "TS⋆ LOC" and "JS LOC" are lines of code in TS⋆ and JavaScript respectively.

Bench	TS⋆ LOC	JS LOC
Point	27	200
QueryString	65	269
CrossDomainMsg	91	427
Csrf	77	334
Csrf-whitelist	85	423
FB-API	39	191

In the remainder of this section, we describe how TS⋆ programs are securely deployed on web pages.

5.1 Securely bootstrapping the TS⋆ runtime

The guarantees of TS⋆ depend on `boot.js` being the first script to run on a web page. Many prior works have implicitly assumed that scripts are always executed in the order in which they appear on the page (Jim et al. 2007; Magazinius et al. 2010; Taly et al. 2011), but, as we explain, this is a naïve view. Instead, we develop a standards-based mechanism that ensures that our scripts run first.

We show how to deploy our scripts safely on any website (`W.com`). First, we describe a novel construction that can ensure that a given script executes first on a page. Second, we describe a technique for loading scripts that contain embedded secrets in a way that protects them from malicious websites. Finally, we present some interoperability and performance results for our examples.

In §2.3, we propose to load a script `boot.js` first on a page so that it can keep clean copies of various primitives that we rely on. More generally, we may want to load a series of first-starter scripts that provide reliable copies of useful libraries, such as JSON and XMLHttpRequest, for use by subsequent code.

Suppose a script *s* is lexically the first element in the header of a page located at a URL `U = http://W.com/page.html`, one may expect that it will be guaranteed to run first on any window loaded from `U`. However, this intuition is correct only if the page has not been loaded programmatically from JavaScript by another web page, e.g., within another frame. On a page loaded initially from `U`, the script *s* will indeed run first. Still, a malicious script running later on the page or on a different page with the same ori-

gin `http://W.com` may open a window or frame at `U`, and modify all the essential primitives before *s* begins to run inside the new window frame. This execution order is consistent with the HTML standard (Berjon et al.) and we have confirmed it experimentally on all mainstream browsers.

Hence, any naïve first-starter implementation that relies on lexical ordering of script elements will fail if other scripts on the same origin are allowed to open windows or frames. Indeed, the web browser only provides security guarantees at the granularity of an origin (Barth 2011); finer-grained privilege separation between good and bad scripts within the same origin require application-level mechanisms, such as restricting all untrusted scripts to a sub-language like SES_{light} (Taly et al. 2011); loading them in sandboxed iframes with few privileges (Akhawe et al. 2012); or modifying the browser to allow the first-starter script to certify all other scripts running on the page (Jim et al. 2007).

Rather than restrict the functionality of untrusted scripts, we propose a standards-based mechanism that ensures that our scripts run first. For a given website, we use two distinct origins:

- `http://W.com`, which is used primarily as service origin; it does not serve any resource.
- `http://start.W.com`, serves HTML pages, including scripts compiled from TS⋆, but where the first two `<script>` elements on the page are the following:

  ```
  <script src="http://start.W.com/boot.js"></script>
  <script>document.domain = "W.com"</script>
  ```

  The first-starter scripts could also be inlined on the page. This server may also handle any other requests (such as XMLHttpRequests), but must not serve any HTML page which does not have the above structure.

  Our implementation also captures more general patterns. For instance, we allow arbitrary HTML pages on `W.com` but treat them as untrusted.

The crucial step here is that after `boot.js` has loaded, the page sets `document.domain` to the parent domain `W.com`. This is a standards-based mechanism (Berjon et al., 5.3.1) by which the page voluntarily gives up its rights to the `http://start.W.com/` origin for client-side same-origin access across frames and windows. Instead, it adopts an *effective script origin* of `http://W.com`.

All subsequent scripts on the page are unrestricted except that they can only read or write into frames or windows that have an effective script origin of `http://W.com`, and hence they cannot tamper with pages on `http://start.W.com`, even if such pages are loaded programmatically into other frames or windows. In all other ways, their functionality is unimpeded and requires no expensive translations or messaging protocols, unlike previous approaches.

More generally, by placing other trusted scripts after `boot.js` and before the assignment to `document.domain`, we can run scripts that grab reliable copies of builtin libraries, such as JSON and `XMLHttpRequest`, for use by subsequent code. Furthermore, we add additional protections against network errors and adversaries.

5.2 Loading scripts with embedded secrets

Our first-starter protocol reliably allows `boot.js` to build a trustworthy environment for our compiled scripts. Conversely, we sometimes need a way for scripts to be able to verify that their environment is trustworthy. This is particularly important when compiled scripts contain secret tokens embedded within them, e.g., to authenticate themselves to other servers. Embedding secrets as constants within program text may seem like an elementary mistake, but this is the predominant way of distributing these tokens in a JavaScript setting. Secrets within scripts must first be protected from malicious websites that may try to load our scripts, and second from malicious scripts on our own website `W.com`. In this threat model, many simple countermeasures one may think of are inadequate.

Even if we require a login cookie for authentication before serving the script, a malicious website that an innocent user visits when logged into `W.com` can make a cross-site script request and obtain the script (an attack sometimes called JavaScript hijacking.) If we inline the scripts into our page, malicious scripts can read their source code and obtain the token. Even if they are not inlined but served from `http://start.W.com`, malicious scripts can perform an `XMLHttpRequest` to obtain their source code and then read them. Indeed, these are all methods commonly used by cross-site scripting (XSS) attacks (e.g., the Samy worm) to break token-based security protections on the web.

To protect our scripts from same-origin attackers, we use a third distinct origin to serve our scripts:

- `https://src.W.com`, the secure source server, only serves GET requests for scripts that may embed secret tokens to be shared between the server and the script. It has a separate origin, with CORS disabled, so the same-origin policy forbids XHR to this server, and it returns scripts with content-type text/javascript, so browsers either run it or ignore it, but never leak its source to other origins. (This is another well-established same-origin assumption.)

To protect our scripts against other websites, we need an additional check Every script served from `src.W.com` is prefixed by a condition on the current webpage location, that is, before making any use of its secret token, the script checks that `window.location.href` begins with `http://start.W.com/` For example, the compiled `csrf.js` served from this server has the form:

```
if (window.location.href === "http://start.W.com/") {
(function(){
 var csrfToken = "XYZ..."; /* Session-specific Token */
 var targetOrigin = "start.W.com";
 function authRPC(...){...}
 Q.defineProperty(win,"authRPC",
  Q.wrap(⟪(string,string,string) → string⟫, ⟪un⟫)(authRPC));
})();
}
```

More generally, we can modify the conditional to allow pages of the form `http://start.W.com/*`. This ensures that the script has a reliable `Q` object on that page, introduced by `boot.js`.

Experimentally, we found that checking the current location of a script is quite error-prone. Some scripts try to read `document.domain` (see e.g., OWASP CSRFGuard in 6.1) or `document.location`, others rely on `window.location.href` but then use regular expression or string matching to check it against a target origin. All these techniques lead to attacks because a malicious website could have tampered with its `document` object or with the regular expression libraries. We found and reported such attacks to vendors.

Notably, many browsers allow `document.domain` and `window.location.origin` to be overwritten. Our origin check relies on the `window.location.href` object which is specified as unforgeable in the HTML specification (Berjon et al., 5.2). In practice, however, we found that some browsers incorrectly allow even supposedly unforgeable objects like `window.document` and `window.location` to be shadowed. We have reported these bugs to various browsers and are in discussions about fixes. If the unforgeability of `window.location.href` turns out to be too strong an assumption, we advocate the use of the origin authentication protocol from Bhargavan et al. (2013).

6. Secure web programming with TS⋆

Modern web applications rely heavily on AJAX for responsiveness, so that data can be retrieved on-demand. Hence, major websites such as Google and Facebook provide rich client-side JavaScript APIs, both for their own pages and for third-party websites, which

can be called by scripts to programmatically access user data. Controlling access to user data relies on combinations of cookies, tokens, and credentials to authenticate the user and the page origin.

We illustrate TS⋆on such existing access control patterns, as deployed in popular JavaScript APIs. We focus on client-sided, language-based security, relegating most other details to the online materials (notably a description of same-origin policies, protocol- and browser specific security assumptions, and current attacks against them; see also e.g., Bhargavan et al. 2013).

6.1 OWASP CSRFGuard

We first consider the task of securing client code that performs the `XMLHttpRequest` to the page's server.

Cross-Site Request Forgeries (CSRF). Suppose a website `W` has an API available to JavaScript. Authenticating and authorizing access using cookies ensures that only requests from a logged-in user's browser are accepted. Conversely, if the user has any another websites open in her browser, their scripts also get access to the API, and can thus steal or tamper with the user's data on `W`. Such request forgery attacks are persistently listed in OWASP Top 10 vulnerabilities, and have a serious impact on a variety of websites.

CSRF Tokens. As a countermeasure to CSRF, a website `W` can inject a fresh, session-specific, random token into every page it serves, and only accept requests that include this token. Other websites cannot see `W`'s pages (thanks to the same origin policy) hence cannot forge requests. Additionally, cookies and tokens can be protected while in transit by using the HTTPS protocol. CSRF protections typically focus on forms embedded in pages, but do not protect dynamic AJAX or XMLHttpRequest. (Until recently, it was commonly, and incorrectly, believed that AJAX calls are not vulnerable to CSRF due to a special header, but this protection proved fragile and easily bypassed.)

OWASP CSRFGuard 3 is the most widely-referenced CSRF protection library. As an advanced feature, it provides a token-injection script that transparently protects AJAX calls by intercepting any `XMLHttpRequest`. (Similar protections exist for frameworks like Django and Ruby-On-Rails.) Crucially, the token must be kept secret from other websites, and also from other scripts loaded on the page; otherwise those scripts may use it to directly perform arbitrary requests, or leak it to some other website.

The token injection script contains an embedded CSRF token, which it must protect from other websites, while including it in requests to W. To do this it has to address three challenges: (1) It should not be possible to load and execute the script on an untrusted website; (2) Even on W, the script should not inject tokens on requests meant for third-party websites; (3) It needs direct access to the XmlHttpRequest object so that it can wrap and protect all uses of this object.

The token injection script is meant to transparently protect an arbitrary website, so does not make any assumption about how the website is written, i.e., the website is Un. However, all CSRF protections explicitly assume the absence of malicious scripts, and do not provide any guarantees against (say) XSS attacks. I.e. they assume the website cannot be Un. Can we do better? (Indeed, one of the first tasks of a typical XSS attack is to steal the CSRF token. The most popular example of such an attack is the MySpace samy worm, which propagates by stealing CSRF tokens.)

Attacks. The original, unwrapped version of their code relies on `window.location`, `String.startsWith`, and `XMLHttpRequest`, which can be tampered with by a malicious script. We found several such attacks where a malicious website could load the OWASP CSRFGuard script, forge the location, and trick it into releasing the token; we are in discussion with the author towards more robust designs, such as the TS⋆one proposed here.

CSRFGuard in TS⋆. Following the approach of §2, we migrate to TS⋆ the OWASP proxy that uses the token to provide authentication RPCs to the rest of the library. This small script is typed in TS⋆, guaranteeing that no malicious script that runs alongside (including the rest of the library) can tamper with its execution. Now that we have complete mediation, we may additionally enforce some access control policies,for instance constraining the kind of requests, their number of requests, and their airguments.

The TS⋆proxy, listed below, takes three string arguments: the target URL, the API function name, and the JSON-formatted arguments. It checks that the URL is well-formed and belongs to the current site (to avoid leaking the token to any other site), then it serializes the request as a query string, attaches the token, and makes an AJAX call. Once wrapped, it exports the same interface as before to any (untrusted) scripts loaded on the page. Additionally, it could be directly used by further TS⋆ code.

```
var csrfToken: string = "%GENERATED_TOKEN%"
var targetOrigin: string = "%TARGET_ORIGIN%"
function Rpc(url:string,apifun:string,args:string): any {
  if (String.startsWith(url,String.concat(targetOrigin,"/")) &&
      String.noQueryHash(url)) {
    var m = {method:apifun, args:args, token: csrfToken};
    var request = String.concat("?",QueryString.encode(m));
    var response = xhrGet(String.concat(url,request));}
    return QueryString.decode(response); }
  else return "unauthorized URL"; }
```

The first two lines define string literals, inlined by the server as it generates the script—the TS⋆ compilation process ensures, via lexical scoping, that these two strings are private to this script. The `Rpc` function is our secure replacement for `xhrGet`; which performs the actual `XMLHttpRequest`. Compared with the original JavaScript, it includes a few type annotations, and uses either safe copies of builtin libraries, such as `xhrGet`, or typed TS⋆ libraries, such as `QueryString` (outlined below). Relying on memory isolation and secure loading from TS⋆, a simple (informal) security review of this script lets us conclude that it does not leak `csrfToken`.

Experimentally, we modified the OWASP library, to isolate and protect the few scripts that directly use the token (such as the proxy above) from the rest of the code, which deals with complex formatting and browser-specific extensions, and is kept unchanged and untrusted. The modified library retains its original interface and functionality, with stronger security guarantees, based on strict, type-based isolation of the token. Its code and sample client- and server-side code are available online. To our knowledge, it is the first CSRF library that provides protection from untrusted scripts.

6.2 Facebook API

Taking advantage of Cross-Origin Resource Sharing (CORS), Facebook provides a client-side JavaScript API, so that trusted websites may access their personal data—once the user has opted in. Interestingly, Facebook also provides a "debug-mode" library, with systematic dynamic typechecks similar to those automated by TS⋆, to help programmers catch client-side errors. We focus on two aspects of their large API: the encoding of strings, and cross-domain messaging.

QueryString encoding. We give a TS⋆ implementation of the `QueryString` module (mentioned above) for the REST message format used in the Facebook API.

```
function decode (s:string): any {
  var res: any = {};
  if (s === "") { return res; } else {
      var params: array string = String.split(s,"&");
      for (var k in params) {
          var kv: array string = String.split(params[k],"=");
          res[kv["0"]] = kv["1"];};
      return res;}}
```

(The `encode` function is dual.) Our function illustrates support for arrays. Importantly, this code may be used to parse untrusted messages; our wrapper for `un` to `string` is straightforward—if the argument is already a string, it is just the identity. Hence, one can write efficient TS* that calls `decode` to parse messages received from the adversary; this coding style prevents many parsing pitfalls.

Another TS*sample illustrates the usage of `Rpc` and our typed JSON library (generalizing `QueryString`) to program a higher-level, statically typed API. It shows, for instance, how to program a client-side proxy for the "/me" method of the Facebook API, which retrieves the user profile; this TS* function has the return type:

```
type profile =
  {id: string; email: string; age_range: {min:number}; ... }
```

Cross-Domain Messaging. The Facebook API is meant to run on any website and protects itself from a malicious host by using iframes. For example, if the website calles `FB.login`, the API loads an iframe from `facebook.com` that retrieves the current user's access token and then only sends it to the host website (via `postMessage`) if the host is in a list of authorized origins.

Bhargavan et al. (2013) report attacks on a prior version of this authorization code that were due to typing errors (and have now been patched). We reimplement this code in TS* and show how programmers can rely on typing to avoid such attacks.

The `checkOrigins` function below is given the current host origin and verifies it against an array of authorized origins. The `proxyMessage` function uses this check to guard the release of the token to the parent (host) website, using a safe copy of the primitive `postMessage` function.

```
function checkOrigins (given:string,expected:array string):bool{
  for (var k in expected) {
    if(given === expected[k]) return true;}
  return false;}
function proxyMessage(host:string,token:any,
                      origins:array string): any {
  if (checkOrigins(host,origins))
    postMessage('parent',token,host);}
```

In a previous version of the Facebook API, `proxyMessage` was accidentally called with an `origins` parameter of type `string`, rather than `array string`. This innocuous type error leads to an attack, because the untyped version of the code succeeds with both strings and arrays, but with different results. To see the core problem, consider a call to `checkOrigins` where `given = "h"` and `expected = "http://W.com"`. The `for` loop goes through `origins` character-by-character, and hence succeeds, when it should not.

In our code, this error is caught statically (if the incorrect call to `proxyMessage` is local) or dynamically (if the call is from another iframe); the check fails in both cases and the token is not leaked.

7. Conclusions and prospects

This paper aims to broaden the scope of gradual typing: not only it is useful for migrating dynamically type-safe code to more structured statically typed code, it is also useful for moving from unsafe code, vulnerable to security attacks, to a robust mixture of dynamically and statically type-safe code.

Within the context of JavaScript, we have presented TS*, a language with a gradual type system, a compiler, and runtime support that provides several useful safety and confinement properties. Our preliminary experience suggests that TS* is effective in protecting security-critical scripts from attacks—without safety and confinement, such properties are difficult to obtain for JavaScript, and indeed security for such scripts has previously been thought unobtainable in the presence of cross-site scripts.

Even excluding the adversary, TS* develops a new point in the design space of gradual typing, using an approach based on runtime type information. This has several useful characteristics, including a simple and uniform failure semantics, and its applicability to a language with extensible objects and object identity.

In the future, we plan to develop TS* along several dimensions. On the practical side, we expect to integrate our ideas in an experimental branch of the open source TypeScript compiler, targeting the construction of larger secure libraries. On the theoretical side, we plan to explore the formal certification of our compiler and runtime. We also hope to develop our preliminary ideas on new notions of blame to explain TS*'s runtime failures.

The TS* compiler will be available on the web, at a URL in the supplementary material. Give it a go!

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM ToPLAS*, 13(2):237–268, 1991.

D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *Proceedings of USENIX Security*, 2012.

T. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of POPL*, 2012.

A. Barth. The web origin concept, 2011. IETF RFC6454.

A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of CCS*, 2008.

R. Berjon, T. Leithead, E. Navara, E.D.and O'Conner, and S. Pfeiffer. HTML5. http://www.w3.org/TR/html5/. W3C Candidate Recommendation.

K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of USENIX Security*, 2013.

G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C$^\sharp$. In *Proceedings of ECOOP*, 2010.

G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of OOPSLA*, 1993.

W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of CCS*, 2012.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ICFP*, 2002.

C. Flanagan. Hybrid type checking. In *Proceedings of POPL*, 2006.

C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Proceedings of POPL*, 2013.

A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of CSFW*, 2001.

A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proceedings of ECOOP*, 2010.

A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of ESOP*, 2011.

D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of CSF*, 2012.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 2010.

L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of OOPSLA*, 2011.

T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of WWW*, 2007.

J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Proceedings of NordSec*, 2010.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *Proceedings of ESOP*, 2009.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of ICFP*, 2011.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of PLDI*, 2013.

A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *Proceedings of S&P*,

2011.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of ESOP*, 2009.

A. A formal embedding in JS⋆

A.1 A review of JS⋆ with a partitioned heap model

```
type tag = Inv:tag | Ref:tag | Abs:tag | Un:tag | Stub:tag | Q:tag (* new for TS* *)
```

- ***Inv***: the invariant private heap Let-bound variables and arguments are immutable in f⋆ but are held in heap locations in js⋆. To keep track of these locations, we place them in a logical compartment called the *Inv* heap. A complication that we handle is that these locations are not strictly immutable—JavaScript forces us to pre-allocate locals, requiring a mutation after allocation, and the calling convention also involves implicit effects. Still, we prove that, once set, all the relevant fields of objects in the *Inv* heap never change.
- ***Ref***: the heap of private source references Locations used to represent the translation of f⋆ values of type *ref t* are placed in the *Ref* heap, where an invariant ensures that the content of a *Ref* heap cell, once initialized, always holds a translation of a *t*-typed source value.
- ***Abs***: the abstract heap of function objects Recall that every function in js⋆ is associated with a heap-allocated object whose contents is updated at every function call. We place these unstable locations in the *Abs* heap, and ensure that translated source programs never read or write from these locations, i.e., function objects are abstract.
- ***Un***: the untrusted heap This heap compartment is used to model locations under control of the attacker. Our full-abstraction result relies crucially on a strict heap separation to ensure that locations from the other compartments never leak into the *Un* heap (with one exception, discussed next).
- ***Stub***: the heap of declassified function objects Function objects corresponding to stubs in the `upfun` wrapper are allocated in a compartment of their own.

  For typing embdedded TS⋆, we introduce one new heap fragment:

- ***Q***: invariant singleton heap, shared between TS⋆ and the context.

Concretely, the following definitions set up the ambient signature of js⋆ against which we program and verify JavaScript core runtime functionality.

Monadic computation types. All computations in js⋆ are typed in the Dijkstra monad, *DST*. We recall its signature below.

```
type result a = V : a → result a | E : exn → result a | Err : result a
type DST :: a::⋆ ⇒ ((result a ⇒ heap ⇒ E) ⇒ heap ⇒ E) ⇒ ⋆
val returnDST : x:a → iDST a (Λ p h. HeapInv h ∧ p x h)
val bindDST : DST a wp1
              → (x:a → DST b (wp2 x))
              → DST b (Λp h. wp1 (λ r. r=V _ ⟹ wp2 (V.0 r) p ∧ ...))
```

For enforcing heap invariants, we use an enhancement of *DST*, called *iDST*.

```
type iDST a WP = DST a (WithInv a WP)
where WithInv a WP = Λp.λh.
        HeapInv h ∧ WP (λx h'. p x h' ∧ HeapInv h' ∧ δ h h') h
```

Representation of dynamically typed expressions. The following nest of mutually recursive types defines the representation of JavaScript's dynamically typed values in JS⋆.

```
logic array(string,pr) type obj
and pr = Data : dyn → pr | Accessor : get:dyn → set:dyn → pr
and tobj = TO: ∀p. t:tag → o:obj{p o} → v:tobj{TypeOf v=Object p t}
and loc = TL: ∀p. t:tag → ref (v:tobj{TypeOf v=Object p t})
        → v:loc{TypeOf v=ORef p t}
and dyn = ...
| Bool: bool → d:dyn{TypeOf d=bool}
| Num: float → d:dyn{TypeOf d=float}
| Str: string → d:dyn{TypeOf d=string}
| Obj: l:loc → d:dyn{TypeOf d=TypeOf l}
| Fun: ∀wp. o:dyn{Obj_is l}
        → (this:dyn → args:dyn → iDST dyn (wp o args this))
        → d:dyn{TypeOf d=WP wp}
```

The type *obj* is an abstract type for JavaScript's dictionary-based object, interepreted in the logic as a map from *string* to properties *pr*, where properties can either be *Data*-values, or *Accessor*s.

The type *tobj* associates with every object in the heap an object invariant *p* (instantiated by the constructor of the object), a tag *t* recording the heap compartment in which the object resides—both of these are purely specificational. The object *o* itself is refined by the object invariants, and the refinement on the result recalls both the invariant *p* and the tag *t*. Locations *loc* are heap references to *tobj* values, with the suitable invariant and tag.

Type *dyn* is a refined type dynamic, where the refinement formulas serve to recover information about the values injected into *dyn*. Notably, the refinement of a function is the predicate transformer *wp* that is its logical specification in the *iDST* monad.

Abbreviations and notations. We make use of the following logical functions in the specifications that follow.

Inside (*t*:*tag*)	≜	$t \in \{Ref, Abs\}$
Inside (*d*:*dyn*)	≜	*Inside d.tag*
h[*v*] : *obj*	≜	(*Sel h v.loc.ref*).*obj*
o[*f*] : *dyn*	≜	(*SelObj o f*).*value*
o[[*f*]] : *pr*	≜	*SelObj o f*
h[*v*] ← *o*	≜	*Upd h v.loc.ref o*
o[*f*] ← *v*	≜	*UpdField o f* (*Data v*)
(*Obj l*).*loc*	≜	*l*
(*Fun _ o _*).*loc*	≜	*o.loc*
(*TO _ t _*).*tag*	≜	*t*
(*TL _ t _*).*tag*	≜	*t*
(*Str _*).*tag*	≜	*Inv*
(*Num _*).*tag*	≜	*Inv*
(*Bool _*).*tag*	≜	*Inv*
v.tag	≜	*v.loc.tag* when $v \in \{Obj\ \_, Fun\ \_\ \_\ \_\}$
(*TL _ _ r*).*ref*	≜	*r*
(*Obj l*).*obj*	≜	(*Obj l*)
(*Fun _ o _*).*obj*	≜	*o*
(*TO _ _ o*).*obj*	≜	*o*
(*Data v*).*value*	≜	*v*

A.2 Representation of runtime type information

We use it the datatype *typ* as a logical model of the JS⋆ values that represent TS⋆ types.

```
type typ =
  | String : typ
  | Number : typ
  | Any : typ
  | Un : typ
  | Data : string → typ
  | Arrow : typ → typ → typ
  | Rec : list (string * field) → typ
and field = (bool * typ)
```

We define a heap-predicate, *Rep t o h*, to assert that the contents of *o* in the heap *h* is a representation of the source type *t*.

$$
\begin{array}{lcl}
\textbf{type } \mathit{Rep} & :: & \mathit{typ} \Rightarrow \mathit{obj} \Rightarrow \mathit{heap} \Rightarrow E \\
\textbf{type } \mathit{RepF} & :: & \mathit{field} \Rightarrow \mathit{obj} \Rightarrow \mathit{heap} \Rightarrow E \\
\mathit{Rep\ String\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"string"} \\
\mathit{Rep\ Number\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"number"} \\
\mathit{Rep\ Any\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"Any"} \\
\mathit{Rep\ Un\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"Un"} \\
\mathit{Rep\ (Data\ s)\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"data"} \wedge o[\texttt{"name"}]{=}\mathit{Str\ s} \\
\mathit{Rep\ (Arrow\ t1\ t2)\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"arrow"} \\
 & \wedge & \mathit{Rep\ t1}\ (h[o[\texttt{"arg"}]])\ h \\
 & \wedge & \mathit{Rep\ t2}\ (h[o[\texttt{"ret"}]])\ h \\
\mathit{Rep\ (Rec\ fs)\ o\ h} & = & o[\texttt{"t"}]{=}\mathit{Str}\ \texttt{"record"} \\
 & \wedge & \forall f{,}t \in \mathit{fs}.\ f \in \mathit{dom}\ o \wedge \mathit{RepF}\ (f{,}t)\ h[o[f]]\ h \\
 & \wedge & \mathit{AsSet}\ o[\texttt{"fields"}]\ h = \{\, f \mid (f{,}\_) \in \mathit{fs} \} \\
\mathit{RepF\ (m{,}t)\ o\ h} & = & o[\texttt{"mut"}]{=}\mathit{Bool\ m} \wedge \mathit{RepF\ t}\ h[o[\texttt{"t"}]]\ h
\end{array}
$$

Lemma 2 (Unique representation of types)**.**
$\forall t\ t'\ o\ h.\ \mathit{Rep\ t\ o\ h} \wedge \mathit{Rep\ t'\ o\ h} \Longrightarrow t{=}t'$

Proof. By induction on the structure of t. □

A.3 Translation of types

All values of source TS* type t are represented as JS* of type *dyn*. To recover the precision of source types in JS* we introduce type indexed heap predicates ψ_t, where $\psi_t\ d\ h$ states that the value d:dyn is the translation of a source value of type t in the heap h. Since a source value allocated at the type $\{f :^a \mathit{number}\}$, may evolve to become a value of type $\{f :^a \mathit{number}, g :^a \mathit{number}\}$, we require a type-indexed heap-predicate.

Heap predicates for source types. Next, we show the definition of $\psi_t d\ h$, a predicate asserting that in the heap h, the value d is the translation of some source value of type t.

(In the main paper text, ψ_t is written as $[\![t]\!]$, $\psi' t$ as *Tagged t*, and finally ψ''_t as *Is t*.)

$$
\begin{array}{lcl}
\psi_{\mathit{string}}\ d\ h & = & \mathit{TypeOf\ d{=}string} \\
\psi_{\mathit{number}}\ d\ h & = & \mathit{TypeOf\ d{=}float} \\
\psi_{\mathsf{un}}\ d\ h & = & \mathit{IsUn\ d} \\
\psi_t\ d\ h & = & \exists u <: t.\, \psi'_u\ d\ h \wedge \mathit{Initialized}\ h[d] \\
\psi'_t\ d\ h & = & \psi''_t\ d\ h\ \ \wedge\ \ \mathit{Rep\ t}\ h[h[d][\texttt{"rtti"}]]\ h \\
\psi''_{\mathsf{any}}\ d\ h & = & \mathit{Primitive\ d} \vee \mathit{Fun\_is\ d} \\
 & \vee & (\mathit{TypeOf\ d{=}ORef}\ \iota\ \mathit{Ref}\ \ \wedge\ \ \mathit{restAny}\ \{\}\ d\ h) \\
\psi''_T\ d\ h & = & \mathit{TypeOf\ d{=}ORef}\ \iota\ \mathit{Ref} \wedge \bigvee_{D_{\bar{t}\to T}} h[d][\texttt{"c"}]{=}\mathit{Str}\ \texttt{"D"} \\
 & \wedge & \wedge_i\ \psi_{t_i}\ (h[d][i])\ h \wedge \mathit{restAny}\ \{1..n\}\ d\ h \\
\psi''_{\bar{f}:\bar{a}\bar{t}}\ d\ h & = & \mathit{TypeOf\ d{=}ORef}\ \iota\ \mathit{Ref} \\
 & \wedge & \forall i.\ i \in \mathit{dom}\ h[d] \wedge \psi_{t_i}\ h[d][f]\ h \wedge \mathit{restAny}\ \overline{f}\ d\ h \\
\mathit{restAny\ fs\ d\ h} & = & \forall f{\in}\mathit{dom}(h[d]) \backslash \mathit{fs}.\ \neg \mathit{Reserved\ f} \Longrightarrow \psi_{\mathsf{any}}\ h[d][f]\ h \\
\psi''_{t_1\to t_2}\ d\ h & = & \mathit{TypeOf\ d} = \mathit{WP}\ \psi_{t_1}\ \psi_{t_2} \wedge \mathit{WithCode\ d} \\
\mathit{WP}\ \psi_{t_1}\ \psi_{t_2} & = & \lambda o\ \mathit{args\ this}.\Lambda p.\lambda h. \\
 & & \mathit{Initialized}\ h[\mathit{args}] \wedge \psi_{t_1}\ h[\mathit{args}][\texttt{"0"}]\ h \\
 & & \wedge \forall r\ h'\ Z.\ (\mathit{ResultIs\ r}\ (\lambda d.\ \psi_{t_2}\ d\ h') \\
 & & \qquad \wedge (\mathit{LocalsOK\ Z\ h} \Longrightarrow \mathit{LocalsOK\ Z\ h'})) \\
 & & \Longrightarrow p\ r\ h' \\
\mathit{WithCode\ d} & = & \mathit{TypeOf\ d.obj{=}ORef}\ (\lambda o.\ \mathit{Initialized\ o} \\
 & & \qquad \Longrightarrow o[\texttt{"@code"}]{=}d)\ \mathit{Abs} \\
\mathit{Reserved\ f} & = & \mathit{Internal\ f} \vee f{=}\texttt{"rtti"}
\end{array}
$$

The case of $\psi_{t\to t'}$ (the translation of source function types) is most interesting. We require the type of the translated value to be function with a predicate transformer $\mathit{WP}\ \psi_t\ \psi_{t'}$ corresponding to the source type—that argument to the function must be initialized and must contain (at index "0") a value satisfying the ψ_t, the heap-predicate for the argument; and if the function returns normally, it produces a value satisfying the heap predicate for the result type. Additionally, $\phi_{t\to t'}$ requires the function object, once initialized, to always contain the translation of the source type in its "rtti" field.

Lemma 3 (ψ-predicates respect subtyping)**.**
$\forall d, h, t, u.t <: u\ \wedge\ \psi_t\ d\ h \Longrightarrow \psi_u\ d\ h$

Proof. By induction on the structure of t, and then induction on the structure of the subtyping judgment. □

Lemma 4 (ψ''-predicates respect Any-subtyping)**.**
$\forall d, h, t.t <: \mathit{Any}\ \wedge\ \psi''_t\ d\ h \Longrightarrow \psi''_{\mathit{Any}}\ d\ h$

Proof. By induction on the structure of t. □

Lemma 5 (ψ-predicates rely only on the inside heaps)**.**
$\forall h\ h'.\ (\forall x.\ \mathit{Inside\ x} \Longrightarrow h[x]{=}h'[x]) \Longrightarrow \forall t\ y.\ \psi_t\ y\ h \Longleftrightarrow \psi_t\ y\ h'$

Proof. By induction on the structure of t. □

Lemma 6 (Uniqueness of ψ')**.**
$\forall t\ u\ v\ h.\ \psi'_t\ v\ h \wedge \psi'_u\ v\ h \Longrightarrow t{=}u$

Proof. Follows from Lemma 2 □

Lemma 7 (ψ' subtype ψ)**.**
$\forall t\ u\ v\ h.\ \psi_t\ v\ h \wedge \psi'_u\ v\ h \Longrightarrow u <: t$

Proof. By unfolding and using Lemma 6. □

Lemma 8 (Function values have arrow tags)**.**
$\forall d\ t\ h.\ (\mathit{Fun\_is}\ d \wedge \psi'_t\ d\ h) \Longrightarrow \exists u_1, u_2.\ t{=}u_1 \to u_2.$

Proof. Case analysis on t (proof by contradiction). □

A.4 Heap invariants

***Un* values.** The predicate *IsUn v* defines when the value v could have been produced by, or can be given to, the context. *Un* values include primitives; references to objects in the *Un* heap; *Stub* objects that have been yielded to the context; or the immutable Q object.

$$
\begin{array}{l}
\mathit{IsUn\ x} \triangleq \mathit{TypeOf\ x} \in \{\mathit{bool}, \mathit{string}, \mathit{float}\} \\
\qquad \vee\ x.\mathit{tag} = \mathit{Un} \\
\qquad \vee\ (x.\mathit{tag} = \mathit{Stub} \wedge \texttt{"@declassified"} \in \mathit{dom\ x}) \\
\qquad \vee\ x.\mathit{tag} = Q \\
\mathit{un} \triangleq x{:}\mathit{dyn}\{\mathit{IsUn\ x}\}
\end{array}
$$

As we will see shortly, it is convernient to lift the notion of *Un* to properties (*IsUnProperty*), objects (*IsUnObj*), and functions (*Un2Un*).

$$
\begin{array}{l}
\mathit{IsUnProperty}\ (\mathit{Data\ x}) \triangleq \mathit{IsUn\ x} \\
\mathit{IsUnProperty}\ (\mathit{Accessor}\ (g, s)) \triangleq \mathit{IsUn\ g} \wedge \mathit{IsUn\ s} \\
\mathit{IsUnObj\ o} \triangleq \forall f.\ f \in \mathit{dom\ o} \Longrightarrow \mathit{IsUnProperty\ o}[[f]] \\
\mathit{Un2Un} \triangleq \lambda o\ \mathit{args\ this\ post\ h}.\ \mathit{IsUn\ this} \wedge \mathit{IsUn\ args} \wedge \\
\qquad\qquad (\forall\ r\ h'.\ \mathit{Result\ r\ IsUn} \Longrightarrow \mathit{post\ r\ h'})
\end{array}
$$

The Q compartment. The predicate *IsQ o* on an object establishes that o's prototype points to the singleton heap compartment, Q. The predicate *QSpec* gives types to each field of `Q.prototype`.

$$
\begin{array}{lll}
\mathit{IsQ}\ o\ h & \triangleq & \mathit{TypeOf}\ o[\texttt{"@proto"}]{=}\mathit{ORef}\ \mathit{QSpec}\ Q \\
 & \wedge & \texttt{"fields"} \in \mathit{dom}\ o \\
 & \wedge & \forall f \in \mathit{AsSet}\ o[\texttt{"fields"}].\ f \in \mathit{dom}\ o \\
 & \wedge & \forall f \in \mathit{dom}\ o.\ f{=}\texttt{"fields"} \vee \mathit{Internal}\ f \vee f{=}\texttt{"rtti"} \\
 & & \quad \vee f \in \mathit{AsSet}\ (o[\texttt{"fields"}])\ h \\
\mathit{QSpec}\ o & = & \mathit{TypeOf}\ o[\texttt{"defineProperty"}]{=}\mathit{ORef}\ \mathit{defineProperty}\ Q \\
 & \wedge & \mathit{TypeOf}\ o[\texttt{"hasOwnProperty"}]{=}\mathit{ORef}\ \mathit{hasOwnProperty}\ Q \\
 & \wedge & \ldots \\
 & \wedge & \mathit{TypeOf}\ o[\texttt{"wrap"}]{=}\mathit{ORef}\ \mathit{wrap}\ Q
\end{array}
$$

***HeapInv*, the global heap invariant.** Our main heap invariant contains 5 clauses, shown below. Clause (1) asserts that *Un* all values reachable from *Un* values are also *Un*. Clause (2) asserts that once a *Stub* object has been declassified that all its components can safely be yielded to the context. Both these clauses are not TS*-specific, and are inherited from our prior work. Clause (3) asserts that every reference to the Q-compartment has the type of `Q.prototype`. Clause (4) asserts that every field of an object in the *Inv* compartment is initialized. Clause (5) asserts that every object in the *Ref* ccompartment is an instance of Q and all its fields are initialized. Clause (6) asserts that for every initialized object o in the inner heap, the contents of o are safely described by the type represented by the `"rtti"` field of o.

$$
\begin{array}{lll}
\mathit{HeapInv}\ h & \triangleq & \forall x.\ x \in \mathit{dom}\ h \Longrightarrow \\
(1) & & (\mathit{IsUn}\ x \Longrightarrow \mathit{IsUnObj}\ h[x]) \\
(2) & \wedge & (x.\mathit{tag}{=}\mathit{Stub} \wedge \texttt{"@declassified"} \in \mathit{dom}\ h[x] \\
 & & \Longrightarrow \mathit{IsUnObj}\ h[l]) \\
(3) & \wedge & (x.\mathit{tag}{=}Q \Longrightarrow \mathit{Fun\_is}\ x \vee \\
 & & \mathit{TypeOf}\ h[x]{=}\mathit{ORef}\ \mathit{QSpec}\ Q) \\
(4) & \wedge & (x.\mathit{tag}{=}\mathit{Inv} \Longrightarrow \mathit{InitFields}\ h[x]) \\
(5) & \wedge & (x.\mathit{tag}{=}\mathit{Ref} \Longrightarrow \mathit{Obj\_is}\ x \wedge \mathit{IsQ}\ h[x] \wedge \\
 & & \mathit{InitFields}\ h[x]) \\
(6) & \wedge & (x.\mathit{tag}{=}\mathit{Ref} \wedge \mathit{InitializedV}\ x\ h \wedge \\
 & & \texttt{"rtti"} \in \mathit{dom}(h[x])) \Longrightarrow\ \mathit{Typed}\ x\ h \\
(7) & \wedge & x.\mathit{tag}{=}\mathit{Abs} \wedge \mathit{InitializedV}\ x\ h \Longrightarrow \\
 & & \mathit{Typed}\ (h[x][\texttt{"@code"}])\ h
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Typed}\ d\ h & \triangleq & \exists t.\ \psi'_t\ d\ h \\
\mathit{Interp}\ \mathit{rtti}\ d\ h & \triangleq & \exists t.\ \mathit{Rep}\ t\ \mathit{rtti}\ h \wedge\ \psi''_t\ d\ h \\
\mathit{InitializedV}\ d\ h & \triangleq & \mathit{Primitive}\ d \vee \mathit{Initialized}\ h[d] \\
\mathit{Initialized}\ o & \triangleq & o.\mathit{tag} \in \{\mathit{Inv},\mathit{Ref},\mathit{Abs}\} \Longrightarrow \texttt{"@init"} \in \mathit{dom}\ o \\
 & \vee & (o.\mathit{tag} = Q \vee \mathit{IsUn}\ o) \\
\mathit{InitFields}\ o & \triangleq & \forall f.\ f \in \mathit{dom}\ o \wedge \neg\ \mathit{Internal}\ f \\
 & & \Longrightarrow \mathit{Data\_is}\ o[[f]] \wedge \mathit{InitializedV}\ o[f]\ h
\end{array}
$$

δ, the heap evolution invariant. We also constrain how the heap evolves using the 2-state invariant δ, which has 4 clauses. Clause (1) ensures that no heap objects are deallocated (hiding the details of garbage collection). Clause (2) ensures that objects in the *Inv* compartment never change after initialization. Clause (3) is most specific to TS* and ensures that after initialization, the `"rtti"` for an object in the inner heap evolves only "downward" in the subtyping hierarchy. Clause (4) ensures that the contents of RTTI objects never change.

$$
\begin{array}{l}
\delta\ h0\ h1 \triangleq \\
(1)\ (\forall\ l \in \mathit{dom}\ h0.\ l \in \mathit{dom}\ h1 \wedge (\mathit{Initialized}\ h0[l] \Longrightarrow \mathit{Initialized}\ h1[l]) \\
(2) \wedge (\forall\ l \in \mathit{dom}\ h0.\ l.\mathit{tag}{=}\mathit{Inv} \wedge \mathit{Initialized}\ h0[l] \\
\qquad \Longrightarrow (\forall\ f \in \mathit{dom}\ h0[l].\ \mathit{Internal}\ f \vee h0[l][f]{=}h1[l][f])) \\
(3) \wedge (\forall\ l \in \mathit{dom}\ h0.\ \mathit{Inside}\ l \wedge \mathit{Initialized}\ h0[l] \\
\qquad \Longrightarrow (\forall\ t_0, t_1.\ \mathit{Rep}\ t_0\ h0[l][\texttt{"rtti"}]\ h0 \wedge \mathit{Rep}\ t_1\ h1[l][\texttt{"rtti"}]\ h1 \\
\qquad\quad \Longrightarrow t_1 <: t_0)) \\
(4) \wedge (\forall\ o, t.\ \mathit{Rep}\ t\ o\ h0 \Longrightarrow \mathit{Rep}\ t\ o\ h1)
\end{array}
$$

Lemma 9 (δ preserves ψ)**.**
$$
\begin{array}{l}
\forall v\ h0\ h1\ t.\ \mathit{HeapInv}\ h0 \\
\qquad \Longrightarrow \mathit{HeapInv}\ h1 \\
\qquad \Longrightarrow \delta\ h0\ h1 \\
\qquad \Longrightarrow \psi_t\ v\ h0 \\
\qquad \Longrightarrow \psi_t\ v\ h1
\end{array}
$$

.

Lemma 10 (δ preserves ψ' for functions)**.**
$$
\begin{array}{l}
\forall t, u, f, h, h'.\ \psi'_{t\to u}\ f\ h \\
\qquad \Longrightarrow \delta\ h\ h' \\
\qquad \Longrightarrow \psi'_{t\to u}\ f\ h'
\end{array}
$$

Proof. Proof: Use Lemma 9, and then equality on WP and transitivity.Technically, we require WP t1 t2 to be injective in t1, t2. That's easily established by inspection. □

A.5 The *JSVerify* API (partial)

The basic API to each heap compartment is unchanged from JS*. However, we provide a new interface to objects in the *Ref* heap, reflecting the fact that the are all Q objects. We also provide an interface to the new Q-compartment. The API functions are shown in Figure 3, Figure 4, and Figure 5.

Definition 11 (LocalsOK)**.** *We write LocalsOK $\overline{x{:}t}$ h for the following formula, i.e., locals are pairwise distinct, have a "0" field, and satisfy the heap invariant, but are yet to be initialized.*

$$
\begin{array}{l}
\forall y, z \in \overline{x{:}t}.\ y.\mathit{loc.ref} \neq z.\mathit{loc.ref} \\
\bigwedge_{y \in \overline{x:t}}\ y.\mathit{tag} = \mathit{Inv} \wedge y.\mathit{loc.ref} \in \mathit{dom}\ h \wedge\ \texttt{"0"} \in \mathit{dom}\ h[y] \wedge \mathit{not}(\mathit{Initialized}\ h[y])
\end{array}
$$

Notation: We write: $S;\Gamma \vdash \{\mathit{Pre}\}\ e\ \{\mathit{Post}\}$
$\mathit{Post} :: \mathit{heap} \Rightarrow \mathit{dyn} \Rightarrow \mathit{heap} \Rightarrow E$
and $\mathit{Pre} : \mathit{heap} \Rightarrow E$,
for $S;\Gamma \vdash e : \mathit{iDST}\ \mathit{dyn}\ \psi$
and $S;\Gamma, h : \mathit{heap} \vDash \mathit{Pre}\ h \Longrightarrow\ \mathit{WithInv}\ \mathit{dyn}\ \psi\ (\mathit{Post}\ h)\ h$.

Notation: We write: $\Gamma \vdash Z.e\ \{\mathit{Post}\}$ for
JSVerify$;\Gamma, \Gamma_Z \vdash \{\mathit{LocalsOK}(Z)\}\ e\ \{\mathit{Post}\}$, where

From $\Gamma \vdash Z.e\ \{\mathit{Post}\}$ we have $\Gamma \vdash Z'.e\ \{\mathit{Post}\}$, for $Z' \supseteq Z$.

Definition 12 (Local environment)**.** *Given $\mathit{locals}(e) = \overline{x_i}$, a set of top-level let-bound variable bindings in e, we write $\Gamma_{\mathit{locals}(e)}$ for the environment $\overline{x_i{:}\mathit{dyn}\{\mathit{TypeOf}\ x_i = \mathit{ORef}\ \varphi\ \mathit{Inv}\}}$.*

Theorem 2 (Type preservation (strengthened IH))**.**
If $\Gamma \vdash_f e : t\ \rightsquigarrow\ e'$,
for any $Z \supseteq \mathit{locals}(e)$,
$\Gamma' = [\![\Gamma]\!], f{:}\mathit{dyn}, \Gamma_Z$,
$\Gamma' \vdash \mathit{locals}(e).e'\ \{\mathit{Post}\ (Z \setminus \mathit{locals}(e))\ t\}$.
where

$$
\begin{array}{l}
\mathit{Post}\ X\ t = \\
\lambda h\ x\ h'.\ \mathit{ResultIs}\ x\ (\lambda d.\ \psi_t\ d\ h') \wedge \mathit{LocalsOK}\ X\ h \Longrightarrow \mathit{LocalsOK}\ X\ h'
\end{array}
$$

Proof: *We proceed by induction on the translation judgment. We start with the most interesting cases: (T-Let) and (T-Abs), which manipulate local variables, then (T-x), etc.*

$$
\begin{array}{lcl}
\varphi & = & \lambda flds.\texttt{"0"} \in dom\ flds \\
\iota & = & \lambda\_.\ True \\
locals(^g\lambda x{:}t_x.e) & = & g \\
locals(v) & = & \cdot \text{ otherwise} \\
locals(e\ e') & = & locals(e), locals(e') \\
locals(e[e']) & = & locals(e), locals(e')
\end{array}
\qquad
\begin{array}{lcl}
locals(^g e_1[e_2]) & = & g, g_f \\
locals(^g e_1\ e_2) & = & g_f, g_v, g_t, g_{arg}, g_{ret} \\
locals(^g e_1[e_2] = e_3) & = & g, g_f, g_v, g_t \\
\ldots & & \\
locals(\textbf{let} x{:}t_1 = e_1 \textbf{ in } e_2) & = & x, locals(e_2)
\end{array}
$$

$\boxed{\Gamma \vdash_f e : t \rightsquigarrow e'}$

$$\frac{\Gamma(x) = t}{\Gamma \vdash_f x : t \rightsquigarrow readVar\ f\ x}\ \text{(T-X)} \qquad \frac{\Gamma \vdash_f \bar{v} : \bar{t} \rightsquigarrow \bar{e}}{\Gamma \vdash_f D_{\bar{t}\to T}\ \bar{v} : T \rightsquigarrow data\ D\ \bar{e}{:}\bar{t}\ T}\ \text{(T-D)}$$

$$\frac{\Gamma, x{:}t_x \vdash_o e : t \rightsquigarrow e' \qquad \bar{y}{:}\bar{t} = \text{locals}(e) \qquad src \text{ any string constant}}{\Gamma \vdash_f {}^g\lambda x{:}t_x.e : t_x \to t \rightsquigarrow function\ g\ t_x \to t\ src\ \lambda o.\lambda\_.\lambda x.witness(); \textbf{ let } \bar{y} = mkLocalInv\ \varphi\ ()\ \textbf{in}\ e'}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash_f e : t' \to t \rightsquigarrow e_1 \qquad \Gamma \vdash_f e' : t' \rightsquigarrow e_2}{\Gamma \vdash_f e\ e' : t \rightsquigarrow applyT\ t'\ t\ f\ e_1\ \ell_{global}\ (mkInv\ \varphi\ (Init\ \{\texttt{"0"} \mapsto e_2\}))}\ \text{(T-App)}$$

$$\frac{\Gamma \vdash_f e_1 : t_1 \rightsquigarrow e_1' \qquad \Gamma, x{:}t_1 \vdash_f e_2 : t_2 \rightsquigarrow e_2'}{\Gamma \vdash_f \textbf{let } x = e_1 \textbf{ in } e_2 : t_2 \rightsquigarrow setLocal\ f\ x\ e_1'; e_2'}\ \text{(T-Let)} \qquad \frac{\Gamma \vdash_f v : t \rightsquigarrow e \qquad t \in \{bool, \texttt{any}\} \qquad \Gamma \vdash_f e_1 : t \rightsquigarrow e_1' \qquad \Gamma \vdash_f e_2 : t \rightsquigarrow e_2'}{\Gamma \vdash_f \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 : t \rightsquigarrow \textbf{if } asBool\ e \textbf{ then } e_1' \textbf{ else } e_2'}\ \text{(T-If)}$$

$$\frac{\Gamma \vdash_f \bar{e} : \bar{t} \rightsquigarrow \bar{e}' \qquad \{\bar{f} :^{\bar{a}} \bar{t}\} = t \uplus u \qquad S \vdash t <: \texttt{any}}{\Gamma \vdash_g \{f{:}e\} : u \rightsquigarrow record\ g\ \{\bar{f} : \bar{e}\}\ u}\ \text{(T-Rec)} \qquad \frac{\Gamma \vdash e : t \rightsquigarrow e' \qquad t = t' \uplus \{g :^a u\} \qquad \neg Reserved\ g}{\Gamma \vdash_f e.g : u \rightsquigarrow Qselect\ \phi_t\ \phi_u\ f\ e'\ (Str g)}\ \text{(T-Rd)}$$

$$\frac{\Gamma \vdash_f e : t \rightsquigarrow e_1 \qquad t = \_ \uplus \{g :^w u\} \qquad \Gamma \vdash_f e' : u \rightsquigarrow e_2 \qquad \neg Reserved\ g}{\Gamma \vdash_f e.g = e' : u \rightsquigarrow Qupdate\ \phi_t\ f\ e_1\ \texttt{"g"}\ e_2}\ \text{(T-Wr)} \qquad \frac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow e_i'}{\Gamma \vdash_f {}^g e_1[e_2] \vdash_f : \texttt{any} \rightsquigarrow read\ f\ g\ e_1'\ e_2'}\ \text{(A-Rd)}$$

$$\frac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow s_i}{\Gamma \vdash_f {}^g e_1[e_2] = e_3 : \texttt{any} \rightsquigarrow write\ f\ g\ e_1'\ e_2'\ e_3'}\ \text{(A-Wr)} \qquad \frac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow e_i'}{\Gamma \vdash_f {}^g e_1\ e_2 : \texttt{any} \rightsquigarrow applyAny\ f\ g\ e_1'\ e_2'}\ \text{(A-App)}$$

$$\frac{\Gamma \vdash_f e : t' \rightsquigarrow e' \qquad t' \sim t}{\Gamma \vdash \langle isTag\ t\rangle e : \texttt{bool} \rightsquigarrow QIsTag\ f\ (rtti\ f\ t')\ (rtti\ f\ t)\ e'}\ \text{(A-IsT)} \qquad \frac{\Gamma \vdash_f e : t' \rightsquigarrow e' \qquad t' \sim t \qquad t' \neq \texttt{un}}{\Gamma \vdash_f \langle canTag\ t\rangle e \rightsquigarrow QCanTag\ f\ (rtti\ f\ t)\ (rtti\ f\ t')\ e'}\ \text{(A-CanT)}$$

$$\frac{\Gamma \vdash_f e : t' \rightsquigarrow e' \qquad t' \sim t}{\Gamma \vdash_f \langle canWrap\ t\rangle e : \texttt{bool} \rightsquigarrow QCanWrap\ f\ (rtti\ f\ t)\ (rtti\ f\ t')\ e'}\ \text{A-CanW} \qquad \frac{\Gamma \vdash_f e : t' \rightsquigarrow e' \qquad t' \sim t \qquad t' \neq \texttt{un}}{\Gamma \vdash_f \langle setTag\ t\rangle e : t \rightsquigarrow QSetTag\ f\ (rtti\ f\ t')\ (rtti\ f\ t)\ e'}\ \text{(A-SetTag)}$$

$$\frac{\Gamma \vdash_f e : t' \rightsquigarrow e' \qquad t' \sim t}{\Gamma \vdash_f \langle wrap\ t\rangle e : t \rightsquigarrow QWrap\ f\ (rtti\ f\ t')\ (rtti\ f\ t)\ e'}\ \text{(A-Wrap)} \qquad \frac{\Gamma \vdash_f e : t \rightsquigarrow e' \qquad S \vdash t <: t'}{\Gamma \vdash_f e : t' \rightsquigarrow e'}\ \text{(T-Sub)}$$

$$
\begin{array}{lcl}
rtti\ f\ T & = & QData\ f\ (Str\ \texttt{"T"}) \\
rtti\ f\ (t_1 \to t_2) & = & QArrow\ f\ (rtti\ f\ t_1)\ (rtti\ f\ t_2) \\
rtti\ f\ (g{:}^a t \uplus u) & = & QAddField\ f\ (rtti\ f\ u)\ \text{"}g\text{"}\ (Bool[\![a]\!])\ (rtti\ f\ t)) \\
\ldots & & \\
function\ g\ (t \to t')\ src\ v & = & setLocal\ f\ g\ (mkFunAbs\ WP_{\psi_t,\psi_{t'}}\ src\ v); \\
 & & QdefineRTTI\ f\ (readVar\ f\ g)\ (Str\ \texttt{"rtti"})\ (rtti\ f\ (t \to t')); (readVar\ f\ g) \\
record\ f\ \bar{g}{:}\bar{e}\ u & = & \textbf{let } l = newQ\ f\ ()\ \textbf{in let } \_ = Qupdate f\ \bar{g}\ \bar{e}\ \textbf{in let } \_ = QsetMeta f\ \texttt{"rtti"}(rtti\ f\ u)\ \textbf{in } l \\
Data\ D\ \bar{e}{:}\bar{t}\ T & = & \textbf{let } l = \textbf{new } Q\ f\ ()\ \textbf{in let } \_ = Qupdate\ \texttt{"c"}\ Str\ \texttt{"D"}\ \textbf{in let } \_ = Qupdate f\ \bar{i}\ \bar{e}\ \textbf{in let } \_ = QsetMeta f\ \texttt{"rtti"}(rtti\ f\ T)\ \textbf{in } l \\
read\ o\ g\ e\ e_f & = & setLocal\ o\ g\ e_1; setLocal\ o\ g_f\ e_f; \\
 & & \textbf{if } opAnd\ (opEq\ (opTypeOf\ (readVar\ o\ g))\ (Str\ \texttt{"object"}))\ \ (QHasField\ o\ (readVar\ o\ g)\ (readVar\ o\ g_f)) \\
 & & \textbf{then } Qselect\ o\ (readVar\ o\ g)\ (readVar\ o\ g_f)\ \textbf{ else } Qdie\ o \\
write\ o\ g\ e\ f\ v & = & setLocal\ o\ g\ e; setLocal\ o\ g_f\ f; setLocal\ g_v\ v; \\
 & & \textbf{if } opTypeOf\ (readVar\ o\ g)(Str\ \texttt{"object"})\ \textbf{then} \\
 & & \ \ setLocal\ o\ g_t\ (QtagOf\ o\ (readVar\ f\ g)) \\
 & & \ \ \textbf{if } Qmutable\ o\ (readVar\ o\ g_t)\ (readVar\ o\ g_f) \\
 & & \ \ \textbf{then } Qupdate\ o\ (readVar\ o\ g)\ (readVar\ o\ g_f) \\
 & & \qquad (QsetFieldTag\ o\ (readVar\ o\ g_t)\ (readVar\ o\ g_f)\ (readVar\ o\ g_v))\ \ \textbf{else } Qdie\ o \\
 & & \textbf{else } Qdie\ o \\
applyAny\ o\ g\ e_f\ e_v & = & setLocal\ o\ g_f\ e_f; setLocal\ o\ g_v\ e_v; setLocal\ o\ g_t\ (QtagOf\ o\ (readVar\ o\ g_f)) \\
 & & \textbf{if } opTypeOf\ (readVar\ o\ g_f)(Str\ \texttt{"function"})\ \textbf{then} \\
 & & setLocal\ o\ g_{arg}\ (Qselect\ o\ (readVar\ o\ g_t)\ (Str\ \texttt{"arg"})); \\
 & & setLocal\ o\ g_{ret}\ (Qselect\ o\ (readVar\ o\ g_t)\ (Str\ \texttt{"ret"})); \\
 & & \textbf{let } targ, tret = parseRep\ (readVar\ o\ g_{arg}), parseRep\ (readVar\ o\ g_{ret})\ \textbf{in} \\
 & & \textbf{let } args = (mkInv\ \varphi\ Init\ \{\texttt{"0"} \mapsto (QsetTag\ o\ (readVar\ o\ g_{arg})\ (readVar\ o\ g_v))\})\ \textbf{in} \\
 & & applyAny\ targ\ tret\ o\ (readVar\ o\ g_f)\ \ell_{global}\ args \\
 & & \textbf{else } Qdie\ o
\end{array}
$$

$\boxed{[\![\Gamma]\!]}$ Translation of environments

$$
\begin{array}{lcl}
[\![\cdot]\!] & = & \cdot \\
[\![\Gamma, x{:}t]\!] & = & [\![\Gamma]\!], x{:}[\![t]\!] \\
[\![t]\!] & = & x{:}dyn\{TypeOf\ x = ORef\ \varphi\ Inv \wedge \exists h.\ Witness\ h \wedge Initialized\ h[x] \wedge \psi_t\ h[x][\texttt{"0"}]\ h\}
\end{array}
$$

Figure 6. Formal typed translation from TS* to JS*

```
val newQ: caller:dyn
  →unit
  →iDST dyn (fun post h ⇒
      ∀l h′ l′. l ∈ dom(h) ⟹ l ∈ dom(h′) ∧ h′[l] = h[l]
          ∧ l′ ∉ dom(h) ∧ InitializedV l′ h′ ∧ IsQ h′[l′] h′
          ∧ AsSet (h′[l′]["fields"]) = {} ⟹ post (V l′) h′)

val Qselect: caller:dyn
  →l:dyn
  →f:dyn{f=Str s ∧ ¬Reserved s}
  →iDST dyn (fun post h ⇒
      TypeOf l = ORef ι Ref ∧ InitializedV l h ∧ f ∈ dom(h[l])
      ∧ post (V h[l][f]) h)

val Qupdate: caller:dyn
  →l:dyn
  →f:dyn{f=Str s ∧ ¬Reserved s}
  →v:dyn
  →iDST dyn (fun post h ⇒
      TypeOf l = ORef ι Ref ∧ InitializedV l h ∧ InitializedV v h ∧
      let o = h[l][f] ← v; h′ = h[l] ← o in
      ("rtti" ∈ dom(o) ⟹ Interp o["rtti"] l h′) ∧ post (V v) h′)

val QsetMeta: caller:dyn
  →l:dyn
  →f:string{f="rtti"}
  →rtti:dyn
  →iDST dyn (fun post h ⇒
      TypeOf l = ORef ι Ref ∧ InitializedV l h ∧ InitializedV rtti h ∧
      let h′ = h[l] ← (h[l]["rtti"] ← v) in
      Interp h′[rtti] l h′ ∧ post (V rtti) h′)

val QdefineRTTI: caller:dyn
  →l:dyn
  →f:string{f="rtti"}
  →rtti:dyn
  →iDST dyn (fun post h ⇒
      l.tag=Abs ∧ InitializedV l h ∧ InitializedV rtti h ∧
      let h′ = h[l] ← (h[l]["rtti"] ← v) in
      Interp rtti l h′ ∧ post (V rtti) h′)

val QtagOf: caller:dyn
  →l:dyn
  →iDST dyn (fun post h ⇒
      InitializedV l h
      ∧ (Primitive l ⟹ post (V [[TypeOf l]]) h)
      ∧ (Inside l ⟹ ("rtti" ∈ dom(h[l]) ∧ post (V h[l]["rtti"]) h)))

val QsetTag: caller:dyn
  →src:dyn
  →tgt:dyn
  →l:dyn
  →iDST dyn (fun post h ⇒
      InitializedV src h ∧ InitializedV tgt h ∧
      ∃t,u. Rep u src h ∧ Rep t tgt h ∧ ψu l h
          ∧ (∀ r h′. ResultIs r (λ d. ψt d h′) ⟹ post r h′))
```

Figure 3. API to Q objects in heap

```
val QhasField: caller:dyn
  →l:dyn
  →f:dyn
  →iDST Bool (fun post h ⇒
  TypeOf l = ORef ι Ref ∧ InitializedV l h ∧
  (TypeOf f=string ∧ ¬Reserved f ⟹ post (AsBool(f ∈ dom(h[l]))) h))

val Qmutable: caller:dyn
  →rtti:dyn
  →f:dyn
  →iDST dyn (fun post h ⇒
      InitializedV rtti h ∧ ∃t. Rep t h[rtti] h ∧
      post AsBool(TypeOf f=string ∧ ¬Reserved f ∧
      (t=Any ∨ t={f:w _} ⊎ _ ∨ t={ḡ:_}, f ∉ ḡ))) h

val QsetFieldTag: caller:dyn
  →rtti:dyn
  →f:dyn
  →v:dyn
  →iDST dyn (fun post h ⇒
      InitializedV rtti h ∧ InitializedV v h ∧ (Inside v ∨ Primitive v)
      ∧ ∃t. Rep t rtti h ∧ (t=Any ∨ t={f:w _} ⊎ _ ∨ t={ḡ:_}, f ∉ ḡ) ∧
      let u = if t=Any ∨ t={ḡ:_}, f ∉ ḡ then Any else if t={f:w t′} ⊎ _ then t′ in
      (∀ r h′. ResultIs r (λ d. ψ′u d h′) ⟹ post r h′))

val Qwrap: caller:dyn
  →src:dyn
  →tgt:dyn
  →l:dyn
  →iDST dyn (fun post h ⇒
      InitializedV src h ∧ InitializedV tgt h ∧
      ∃t,u. Rep u src h ∧ Rep t tgt h ∧ ψu l h
          ∧ (∀ r h′. ResultIs r (λ d. ψt d h′) ⟹ post r h′))

val QcanTag: caller:dyn
  →src:dyn
  →tgt:dyn
  →l:dyn
  →iDST dyn (fun post h ⇒
      InitializedV src h ∧ InitializedV tgt h ∧
      ∃t,u. Rep u src h ∧ Rep t tgt h ∧ ψu l h
          ∧ post (AsBool(ψt l h)) h)

val QcanWrap: caller:dyn
  →src:dyn
  →tgt:dyn
  →l:dyn
  →iDST dyn (fun post h ⇒
      InitializedV src h ∧ InitializedV tgt h ∧
      ∃t,u. Rep u src h ∧ Rep t tgt h ∧ ψu l h
          ∧ post (AsBool(ψt l h)) h)
```

Figure 4. Q API continued

```
val rtti: caller:dyn
  →t:typ
  →iDST dyn (fun post h ⇒
      ∀h′ l′ l. l ∈ dom(h) ⟹ l ∈ dom(h′) ∧ h[l] = h′[l] ∧ l′.ref ∉ dom(h)
          ∧ Rep t h′[l′] h′ ∧ InitializedV l′ h′ ⟹ post l′ h′)

val opTypeOf: v:dyn
  →t:string{t="object" ∨ t="function"}
  →iDST Bool (fun post h ⇒ InitializedV v h ∧ post (AsBool(
    t="object" ⟹ TypeOf v=ORef _ _ ∧ t="function" ⟹
Fun_is v)) h)
```

Figure 5. Other API

Case T-Let:

$$\frac{\Gamma \vdash_f e_1 : t_1 \rightsquigarrow e_1' \qquad \Gamma, x{:}t_1 \vdash_f e_2 : t_2 \rightsquigarrow e_2'}{\Gamma \vdash_f \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 : t_2 \rightsquigarrow e_{rhs}}$$

Where $e_{rhs} = \textbf{let}\ y = e_1'\ \textbf{in}\ e_{rhs}'$

and $e_{rhs}' = \textbf{let}\ \_ = setLocal\ f\ x\ y\ \textbf{in}\ e_2'$

and $X_1 = locals(e_1)$ *and* $X_2 = locals(e_2)$, *and* $X = X_1, x{:}t_1, X_2$

From the IH, we have,
(1) $\forall Z \supseteq X_1.\Gamma_0 \vdash Z.e_1'\ \{Post\ (Z \setminus X_1)\ t_1\}$
(2) $\forall Z \supseteq X_2.\Gamma_0, x{:}[\![t_1]\!] \vdash Z.e_2'\ \{Post (Z \setminus X_2) t_2\}$
where, $\Gamma_0 = [\![\Gamma]\!], f{:}dyn, \Gamma_Z$,
Our goal (G) is:
$\forall Z \supseteq X.\Gamma_0 \vdash Z.e_{rhs}\ \{Post\ (Z \setminus X)\ t_2\}$

We first show (S1):
$\forall Z \supseteq X_2, x.\Gamma_0, y{:}dyn \vdash \{A\}\ e_{rhs}'\ \{Post (Z \setminus X) t_2\}$
for $A = \lambda h0.\ LocalsOK\ Z\ h0 \wedge \psi_{t_1}\ y\ h0 \wedge Initialized\ y\ h0$

We read off the following triple from the spec of setLocal, strengthening the pre-condition:
$\forall Z \supseteq X_2, x.\Gamma_0, y{:}dyn \vdash \{A'\}\ setLocal\ f\ x\ y\ \{B\}$
for $B' = \lambda h0\ \_\ h1.\ LocalsOK\ (Z \setminus \{x\})\ h1 \wedge x \notin X_2$
$\wedge\ h1 = h0[x] \leftarrow Init\ (h0[x][\texttt{"0"}] \leftarrow y)$
$\wedge\ Witness\ h1$
and $A' = \lambda h0.$
$TypeOf\ x = ORef\ \varphi\ Inv$
$\wedge\ LocalsOK\ Z\ h0$
$\wedge\ Initialized\ y$
$\wedge\ \varphi\ (Init\ (h[x][\texttt{"0"}] \leftarrow y))$
Next, we strengthen the pre-condition A' *to A, based on the following observations:*
$\varphi\ (Init\ (h[x][\text{"0"}] \leftarrow y))$ *is a tautology.*
$TypeOf\ x = ORef\ \varphi\ Inv$ *is derivable from* $\Gamma_{x:t_1}$ *the local environment which is included in* Γ_0.

Now, we have (S1.1.1)
$\forall Z \supseteq X_2, x.\Gamma_0, y{:}dyn \vdash \{A\}\ setLocal\ f\ x\ y\ \{B'\}$

Then, we strenthen *the post-condition* B' *to B where* $B =$
$\lambda h0\ ()\ h1.\ B\ h0\ ()\ h1 \wedge \psi_{t_1}\ y\ h1$
To prove this post-condition, we rely on $\psi_{t_1}\ y\ h0$ *in the pre-condition A*
And Lemma 5 (noting that h1 differs from h0 only in Inv locations)
Next, we weaken the context of (2) (to introduce y in the environment), and weaken the post-condition, getting:
(S1.2.1) $\forall Z \supseteq X_2.\Gamma_0, x{:}[\![t_1]\!], y{:}dyn \vdash \{LocalsOK\ Z\}\ e_2'\ \{Post (Z \setminus X) t_2\}$
Note, $x{:}[\![t_1]\!]$ *is the translation of source binding* $x{:}t_1$.
As such, it requires the local variable x in the translation to be initialized with an object satisfying ψ_{t_1}.
But, we can rearrange the refinement, pushing the ψ_{t_1} *clause into the pre-condition leaving just* $\Gamma_{x:t_1}$, *a local environment of the translation, included in the context* Γ_0, *to get (S1.2).*
(S1.2) $\forall Z \supseteq X_2.\Gamma_0, y{:}dyn \vdash \{Pre\}\ e_2'\ \{Post (Z \setminus X) t_2\}$
where Pre is
$\lambda h.\ LocalsOK\ Z\ h \wedge \exists h'.\ Witness\ h' \wedge Initialized\ h'[x] \wedge \psi_{t_1}\ h'[x][\texttt{"0"}]\ h'$

To arrive at (S1), we introduce the quantified variable $Z \supseteq X_2, x$ *into the context.*
Then, we instantiate (S1.1) with Z, and (S1.2) with $Z \setminus \{x\}$.
To compose the triples, we need to show:
$\Gamma_0, y{:}dyn, h0{:}heap, h1{:}heap \models B\ h0\ ()\ h1 \Longrightarrow Pre\ h1$
Inlining:
(H) $Z \supseteq X_2, x$
(a) $\wedge\ x \notin X_2$
(b) $LocalsOK\ (Z \setminus \{x\})\ h1$
(c) $\wedge\ h1 = h0[x] \leftarrow Init\ (h0[x][\texttt{"0"}] \leftarrow y)$
(d) $\wedge\ Witness\ h1$
(e) $\wedge\ \psi_{t_1}\ y\ h1$
$\Longrightarrow$
(0) $Z \setminus x \supseteq X_2$
(1) $LocalsOK\ Z \setminus \{x\}\ h1$
(2) $\wedge\ \exists h'.$
(2.0) $Witness\ h'$
(2.1) $\wedge\ Initialized\ h'[x]$
(2.2) $\wedge\ \psi_{t_1}\ h'[x][\texttt{"0"}]\ h'$

(0) is trivial froM (H) and (a).
(1) is trivial from (b).
(2) We instantiate h' *with h1.*
(2.1) Initialized h1[x] is easy from (c).
(2.2) Trivial from (e).
So, we have (S1).

Next, before deriving the goal, we strengthen the pre-condition of (1) to obtain: $(1.1)\ \ \Gamma_0 \vdash \{Pre_1\}\ e_1'\ \{Q(Z \setminus X_1, t_1)\} \mid X$
Now, to derive the goal, we introduce the quantified variable $Z \supseteq X$ *into the context.*
Then, we instantiate (1) with Z, and (S1) with $Z \setminus X_1$.
After instantiation, to compose (1) and (S1), we need to show:
$\Gamma_0, h0{:}heap, y{:}dyn, h1{:}heap \models\ \ LocalsOk\ Z\ h0\ \wedge\ Post\ (Z \setminus X_1)\ t_1\ h0\ y\ h1 \Longrightarrow A\ h1$
Inlining:
(H) $Z \supseteq X$
(a) $\wedge\ X = X_1, x{:}t_1, X_2$
(b) $LocalsOK\ (Z \setminus X_1)\ h1$
(c) $\wedge\ \psi_{t_1}\ y\ h1$
(d) $\wedge\ Initialized\ y\ h1$
$\Longrightarrow$
(1) $LocalsOK\ Z\ h1 \wedge \psi_{t_1}\ y\ h1 \wedge Initialized\ y\ h1$

Which is trivial.

Case T-X:

$$\frac{\Gamma(x) = t}{\Gamma \vdash_f x : t \rightsquigarrow e'}$$

where $e' = readVar\ f\ x$
Our goal:
$\forall Z.[\![\Gamma]\!], f{:}dyn, \Gamma_Z \vdash Z.e'\ \{Post\ Z\ t\}$
Introducing Z into the context, from the spec of readVar we get:
$[\![\Gamma]\!], f{:}dyn, \Gamma_Z \vdash \{A\}\ e'\ \{Post\ Z\ t\}$
Where
$Ah =$
(1) $TypeOf\ x{=}ORef\ \varphi\ Inv$
(2) $\wedge\ \exists h'.\ Witness\ h' \wedge Initialized\ h'[x]$
(3) $\wedge\ (\forall h'.\ Witness\ h' \Longrightarrow \delta\ h'\ h) \Longrightarrow Initialized\ h[x][\texttt{"0"}]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h$
(4) $\wedge\ LocalsOK\ Z\ h \Longrightarrow LocalsOK\ Z\ h$ *(∗ trivial ∗)*
From $[\![x{:}t]\!]$ *we get (1) and (2).*
For (3), we use Lemma 9.

Case A-App: $\dfrac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow e_i'}{\Gamma \vdash_f {}^g e_1\ e_2 : \texttt{any} \rightsquigarrow e'}$

where $e' = applyAny\ f\ g\ e_1'\ e_2'$

*From the IH on the two premises, **and with** standard reasoning about locals, after the assignments to g_f **and** g_v, we get:*

(IH1) $\psi_{\texttt{any}}$ *h[g_f]["0"] h*
(IH2) $\psi_{\texttt{any}}$ *h[g_v]["0"] h*

Now, we focus on proving just two key triples, The rest is straightforward.

TRIPLE 1.

targ:typ, tret:typ |

{
 HeapInv h
 ∧ *Rep (targ → tret) h[g_f]["rtti"] h*
 ∧ *Rep targ h[g_arg]["0"] h*
 ∧ *Rep tret h[g_ret]["0"] h*
 ∧ ψ'_{*targ → tret*} *h[g_f]["0"] h*
 ∧ *(Inside (read o g_v) ∨ Primitive (read o g_v))*
}

 QsetTag (read o g_arg) (read o g_v)

{*h, d, h'.*
 HeapInv h' ∧ *delta h h'*
 ∧ ψ'_{*targ*} *d h'*
 ∧ ψ'_{*targ → tret*} *h'[g_f]["0"] h'*}

Proof:

*Reading from the spec **of** QsetTag, we get to prove our triple **if**:*

(a) HeapInv h
(b) ∧ *Rep (targ → tret) h[g_f]["rtti"] h*
(c) ∧ *Rep targ h[g_arg]["0"] h*
(d) ∧ *Rep tret h[g_ret]["0"] h*
(e) ∧ ψ'_{*targ → tret*} *h[g_f]["0"] h*
(f) ∧ *(Inside (read o g_v) ∨ Primitive (read o g_v))*

⟹
(1) ∃*t. Rep t (read o g_arg) h*
(2) ∧ *(Inside (read o g_v) ∨ Primitive (read o g_v))*
 ∀*d h'.*
 (g) \$\psi'$_t *d h'* ∧ *HeapInv h'* ∧ *delta h h'*
 ⟹
(3) HeapInv h' ∧ *delta h h'*
(4) ∧ ψ'_{*targ*} *d h'*
(5) ∧ ψ'_{*targ → tret*} *h'[g_f]["0"] h'*

---

*(1) is derivable from (c), witnessing t **with** targ*
*(2) is the same **as** (f)*
(3) is trivial from (g)
(4) is also from (g)
(5) From the delta h h', for the Inv Heap, we know that
 h[g_f]["0"] = h'[g_f]["0"]
 *Then, from (e) **and** Lemma 10 we get (5).*

TRIPLE 2.

targ, tret, h, args |

{*HeapInv h*
 ∧ *TypeOf args =ORef varϕ Inv*
 ∧ *Initialized h[args]*
 ∧ ψ'_{*targ*} *h[args]["0"] h*
 ∧ ψ'_{*targ → tret*} *h[g_f]["0"] h*}

applyAny targ tret o (readVar o g_f) l_global args

{
 h, r, h'.
 ResultIs r (\d. ψ_Any d h')
}

Proof:

*Reading from the spec **of** applyAny, we get to prove our triple **if**:*

(a) HeapInv h
(a.1) ∧ *TypeOf args =ORef varϕ Inv*
(a.2) ∧ *Initialized h[args]*
(b) ∧ ψ'_{*targ*} *h[args]["0"] h*
(c) ∧ ψ'_{*targ → tret*} *h[g_f]["0"] h*
⟹
(1) TypeOf h[g_f]["0"]=WP ψ_{targ} ψ_{tret}
(2) ∧ *Initialized h[args]*
(3) ∧ ψ_{targ} *h[args]["0"] h*
 ∧ ∀*d h'.*
(d) ψ_{tret} *d h'* ∧ *HeapInv h'* ∧ *delta h h'*
 ⟹
(4) $\psi_{\texttt{any}}$ *d h'*

(1) Unfolding (c), we get (1)
(2) From (a.2)
(3) Unfolding ψ_{targ} ***in** the goal, we get an existential,*
 ***and** we instatiate the the bound var to targ **and** we use (b)*
(4) We aim to show that tret <:`any`
 *-From (IH1) **and** Lemma 3,*
 we get targ → tret <: Any

 -From inversion on the subtyping judgment, we get tret <: Any

 *-From (d) **and** Lemma 3, we get (4).*

***Case T-App**:*

$$\frac{\Gamma \vdash_{\mathcal{T}} e : targ \to tret \rightsquigarrow e_1 \quad \Gamma \vdash_{\mathcal{T}} e' : targ \rightsquigarrow e_2}{\Gamma \vdash_{\mathcal{T}} e\ e' : tret \rightsquigarrow applyT_{targ\ tret\ f\ e_1\ \ell_{global}}\ (mkInv\ \varphi\ (Init\ \{\texttt{"0"} \mapsto e_2\}))}$$

From IH1, we have:

$\{HeapInv\ h1\}$
e_1
{
f, h2.
(1) ResultIs f $(\lambda\ d.\ \psi_{targ \to tret}\ d\ h2 \wedge Initialized\ d\ h2)$
(2) HeapInv h2 $\wedge$ *DeltaHeap h1 h2*
}

From IH2, we have:

{..}
e_2
{
v, h3.
(3) ResultIs v $(\lambda\ d.\ \psi_{targ}\ d\ h3 \wedge Initialized\ d\ h3)$
(4) HeapInv h3 $\wedge$ *DeltaHeap h2 h3*
}

(Let h4 be the next heap after doing args = (mkInv φ *(Init*$\{$`"0"` $\mid\!\to v\}$*)))*
We want to prove the postcondition

applyT targ tret o f glob args
{
r, h5.
(a) ResultIs r $(\backslash d.\ \psi_{tret}\ d\ h5) \wedge Initialized\ d\ h5$
}

Reading the spec ***of*** *applyT, we need to prove:*

(b) $\psi_{targ \to tret}\ f\ h4$
(c) Initialized h4[args] h4
(d) ψ_{targ} *h4[args][*`"0"`*] h4*
$\forall d\ h5.$
(5) $\psi_{tret}\ d\ h5 \wedge HeapInv\ h5 \wedge DeltaHeap\ h4\ h5$
$\Longrightarrow$
(e) $\psi_{ret}\ d\ h5 \wedge Initialized\ d\ h5$

(b) follows from (1) ***and*** *Lemma 9*
(c) follows from mkInv
(d) follows from (3) ***and*** *Lemma 9*
(e) follows from (5)

Case T-Rec*:*
$$\frac{\Gamma \vdash_f \bar{e} : \bar{t} \rightsquigarrow \bar{e}' \qquad \{\bar{f} :^{\bar{a}} \bar{t}\} = t \uplus u \quad S \vdash t <: \texttt{any}}{\Gamma \vdash_g \{f{:}e\} : u \rightsquigarrow record\ g\ \{\bar{f} : \bar{e}\}\ u}$$

From I.H. we have:

$\{\ HeapInv\ h\ \}$
let *v_i = e_i*
$\{\ h'.$
(a_i) ResultIs v_i $(\lambda\ d.\ \psi_{t_i}\ d\ h') \wedge HeapInv\ h' \wedge DeltaHeap\ h\ h'\ \}$

Let h be the heap now (above results hold ***in*** *h by Lemma 9*

$\{HeapInv\ h\}$
let *l = newQ f ()*
$\{true\}$

Let heap be h1 now, ***and*** *we know from the spec* ***of*** *newQ:*
(b) TypeOf l = ORef ι *Ref*$\}$

For each update field call, to prove the triple:

$\{\ heap\ h1\ \}$
updateQ f l g_i v_i
$\{true\}$

we need to prove (from the spec ***of*** *updateQ):*

(1) TypeOf l = ORef l Ref
(2) Initialized v_i
let *o = h1[l][g_i]* $<$ *v_i* ***in let*** *h2 = h1[l]* $<$ *o*
(3) `"rtti"` $\in dom(o) \Longrightarrow$ *Interp o[*`"rtti"`*] o h2*

(1) follows from (b)
(2) follows from a_i ($\psi_t \Longrightarrow$ *Initialized)*
(3) follows from `"rtti"` $\notin dom(o)$

Let h3 be the heap now, from spec ***of*** *updateQ we know:*

(c)$\forall$ *i. h3[l][g_i] = v_i*

$\{h3\}$
let *rt = rtti f u*
$\{true\}$

No precondition on rtti. Let h4 be the heap, ***then*** *from spec* ***of*** *rtti,*

(d) $l \in dom(h3) \Longrightarrow$ *h3[l] = h4[l]*
(e) Rep u h4[rt] h4
(f) Initialized rt

QsetMeta f l `"rtti"` *rt*
$\{\ h5, r\ .$ *ResultIs r* $(\lambda\ d\ .\ \psi_u\ d\ h5)\ \}$

To prove this, we need to prove the following precondition
(from spec ***of*** *QsetMeta)*

(1) l.tag = Ref
(2) Initialized rt
(g) ***let*** *h5 = h4[l]* $<$ *(h4[l][*`"rtti"`*]* $<$ *v)*
(3) Interp h5[rt] l h5
(4) ψ_u *l h5*

(1) follows from (b)
(2) follows from (f)
(3) expanding Interp:

$\exists t$ *s.t. Rep t h5[rt] h5* $\wedge$ ψ_t'' *l h5*

Choose t = u, ***then*** *we need to prove*

3a. Rep u h5[rt] h5
3b. ψ_u'' *l h5*

3a follows from h5[rt] = h4[rt] ***and*** *(4)* ***in*** *Delta specification*
3b expanding ψ_u''*,*

3b1. l.tag = Ref
3b2. $\forall\{g{:}t\} \in u$*,* $g \in dom(h[l])$ $\wedge$ ψ_t *h5[l][g] h5*
3b3. restAny $\bar{u}$ *l h5*

3b1 follows from (b)
3b2 from (c), (d), ***and*** *(g), each* $g \in dom(h[l])$ $\wedge$
from (a_i) ***and*** *Lemma 9*
3b3 From typing rule, fields not ***in*** *u are subtype* ***of*** *Any,* ***and*** *so,*
from (a_i) ***and*** *Lemma 3.*

Case T-Rd: $$\frac{\Gamma \vdash e : t \rightsquigarrow e' \qquad t = t' \uplus \{g :^{a} u\}}{\Gamma \vdash_f e.g : u \rightsquigarrow Qselect\ f\ e'\ g}$$

From IH, we have:

$\{heap\ h\ .\ HeapInv\ h\}$
let *l =* e'
$\{(a)\ heap\ h1\ .\ \psi_t\ l\ h1 \wedge DealtaHeap\ h\ h1 \wedge HeapInv\ h1\}$

We want to prove:

Qselect f l g
$\{\ r, h2\ .\ ResultIs\ r\ (\lambda\ d.\ \psi_u\ d\ h2)\ \}$

Reading spec ***of*** *Qselect, this requires us to prove (h1 = h2):*
(not ***internal*** *f comes from typing rule)*

(1) TypeOf l = ORef ι *Ref*
(2) Initialized h1[l]
(3) $g \in dom(h1[l])$
(4) ψ_u *h1[l][g] h1*

(1) follows from ψ_t ***where*** *t is a record* ***type****.*

From (a), we know ψ_t *l h1,* ***where*** $t = t' \uplus \{g{:}a\ u\}$

That means, $\exists t1 <: t$*, s.t.* ψ'_{t1} *l h1* ***and*** *(2)*
Since t is a record ***type****, t1 has to be a record* ***type*** *(inversion* ***of*** *subtyping)*
and *must have* $\{g{:}a'\ u'\}$ *s.t. either* $u' = u$ *or* $u' <: u$ *(depending on a).*

Then, ψ''_{t1} *l h1 tells us that (3)* $g \in dom(h1[l])$ ***and*** $\psi_{u'}$ *h1[l][g]*
(4) ***then*** *follows from Lemma 3.*

Case T-Wr: $$\frac{\Gamma \vdash_f e : t \rightsquigarrow e_1 \qquad t = \_ \uplus \{f :^{w} u\} \qquad \Gamma \vdash_f e' : u \rightsquigarrow e_2 \qquad \neg Internal\ f}{\Gamma \vdash_f e.g = e' : u \rightsquigarrow updateQ\ f\ e_1\ \texttt{"f"}\ e_2}$$

From IH1, we have:

$\{h{:}heap\}$
let *l = e1*
$\{(a)\ h1\ .\ \psi_t\ l\ h1\ \}$

From IH2, we have:

let *v = e2*
$\{(b)\ h2\ .\ \psi_u\ v\ h2\}$

We want to prove:

Qupdate f l v
$\{h3\ .\ \psi_t\ l\ h3\}$

From spec ***of*** *Qupdate, this requires us to prove:*

(1) TypeOf l = ORef ι *Ref*
(2) Initialized v
$h3 = h2[l] < (h2[l][g] < v)$
(3) $\texttt{"rtti"} \in dom(h3[l]) \Longrightarrow$ *Interp (h3[l][*$\texttt{"rtti"}$*]) l h3*
(4) ψ_t *l h3*

(1) follows from ψ_t ***where*** *t is a record* ***type***
(2) follows from (b)
(3) expanding Interp h3[l][$\texttt{"rtti"}$*] l h3:*

$\exists t1$ *s.t.*
3a. Rep t1 h3[l][$\texttt{"rtti"}$*] h3[l]*
3b. ψ''_{t1} *l h3*

(a) tells us that ψ_t *l h1* ***and*** *we know* δ *h1 h2,*
so we have ψ_t *l h2.*

This means, $\exists tsub <: t$ *s.t.* ψ'_{tsub} *l h2* $\wedge$ *Rep tsub h2[l][*$\texttt{"rtti"}$*] h2*
We instantiate t1 ***in*** *expansion* ***of*** *Interp* ***in*** *(3) above* ***with*** *tsub*

3a follows from h2[l][$\texttt{"rtti"}$*] = h3[l][*$\texttt{"rtti"}$*]* ***and*** *that* $\delta h2h3$

For 3b, we note that only g field is changed from h2[l] to h3[l]
Since t has $\{g{:}w\ u\}$*, tsub must be a record* ***type and*** *have* $\{g{:}w\ u\}$
So, we need to prove ψ_u *v h3 -follows from (b)* ***and*** *Lemma 9.*

Case A-Rd: $$\frac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow e'_i}{\Gamma \vdash_f {}^{g}e_1[e_2] \vdash_f : \texttt{any} \rightsquigarrow read\ f\ g\ e'_1\ e'_2}$$

From IH1, we have:

$\{heap\ h1\}$
let $l = e1$
$\{(a)\ heap\ h2.\ \psi_{Any}\ l\ h2\}$

$\{heap\ h2\}$
let $g = e2$
$\{(b)\ heap\ h3.\ \psi_{Any}\ l\ g\}$

We want to prove
*(proof is modulo setLocal **and** readVar)*

$\{heap\ h3\}$
$Qselect\ f\ l\ g$
$\{h4\ r\ .\ ResultIs\ r\ (\lambda\ d\ .\ \psi_{Any}\ d\ h4)\}$

*(From specs **of** opTypeOf, QhasField, we see that they **do** not modify heap)*

*From spec **of** Qselect, we get h3=h4 **and** we need to prove:*
(1) TypeOf l = ORef ι Ref
(2) InitializedV l h3
(3) TypeOf g=string ∧ ¬Reserved g
(4) g ∈ dom(h3[l])
(5) $\psi_{Any}\ h3[l][g]\ h3$

*(1) follows from postcondition **of** opTypeOf*
*(2) follows from (a) **and** δ h2 h3.*
*(3) **and** (4) follow from postcondition **of** QhasField*
*(5) From (a) **and** δ h2 h3, we get* $\psi_{Any}\ l\ h3$
From Lemma 4, we get $\psi''_{Any}\ l\ h3$
*restAny clause **in*** ψ''_{Any} *tells us that* $\psi_{Any}\ h3[l][g]\ h3$

***Case A-Wr**:* $\dfrac{\forall i.\Gamma \vdash_f e_i : \texttt{any} \rightsquigarrow s_i}{\Gamma \vdash_f {}^{g} e_1[e_2] = e_3 : \texttt{any} \rightsquigarrow write\ f\ g\ e'_1\ e'_2\ e'_3}$

From IH1:

$\{heap\ h1\}$
let $l = e1$
$\{(a)\ heap\ h2\ .\ \psi_{Any}\ l\ h2\}$

$\{heap\ h2\}$
let $g = e2$
$\{(b)\ heap\ h3\ .\ \psi_{Any}\ g\ h3\}$

$\{heap\ h3\}$
let $v = e3$
$\{(c)\ heap\ h4\ .\ \psi_{Any}\ v\ h4\}$

We want to prove:

$\{heap\ h4\}$
let $v' = QsetFieldTag\ f\ (QtagOf\ f\ l)\ g\ v$
$\{heap\ h5\ ...\ \}$
$Qupdate\ f\ l\ g\ v'$
$\{h6,r.\ ResultIs\ r\ (\lambda\ d.\ \psi_{Any}\ d\ h5)\}$

(Qmutable doesn′t mutate heap)

*It′s easy to see that preconditions **of** QsetFieldTag are satisfied.*

*Reading spec **of** Qupdate, precondition **of** Qupdate is:*

(1) TypeOf g=string ∧ ¬Reserved g
(2) TypeOf l = ORef ι Ref
(3) InitializedV l h5
(4) InitializedV v′ h5
h6 = h5[l] < (h5[l][g] < v′)
(5) Interp h6[l]["rtti"] l h6
(6) $\psi_{Any}\ v'\ h6$

*(1) follows from the postcondition **of** Qmutable*
(2) opTypeOf tells us that l is an ORef _ _,
from (a), we derive it′s ORef ι Ref
(3) from (a) it follows that InitializedV l h2,
from δh2h5, (3) follows
(4) follows similarly
(5) we have: $\psi_{Any}\ l\ h5$*, i.e.* $\exists u <: Any\ .\ \psi'_u\ l\ h5$
i.e. $\psi''_u\ l\ h5$ ***and*** *Rep u h5[h5[l]["rtti"]] h5*
*Claim: Rep u h6[h6[l]["rtti"]] h6, proof by third condition **of** δ*
Claim: $\psi''_u\ l\ h6$
*proof sketch: we have u = Any, u = record **type with** no g, u = {g:t_g} ⊎ _*
in *every case, postcondition **of** QsetTag ensures that v′*
*respects RTTI assumption **of** u on g*
*Thus, choose t=u **in** the expansion **of** Interp*
*(6) When u = Any or u = record **with** no g, (6) immediately follows from*
*postcondition **of** QsetTag. When u has {g:t_g}, from* $\psi_{Any}\ l$*,*
*we know t_g <: Any, **and then** (6) follws from ψsubtyping.*

***Case T-Abs**::*

$$\frac{\Gamma, x{:}t_X \vdash_{\mathcal{D}} e : t \rightsquigarrow e' \qquad \bar{y}{:}\bar{t} = locals(e) \qquad src\ any\ string\ constant}{\Gamma \vdash_f {}^{g} \lambda x{:}t_X.e : t_X \to t \rightsquigarrow function\ g\ t_X \to t\ src\ \lambda o.\lambda\_.\lambda x.witness();\ \textbf{let}\ \bar{y} = mkLocalInv\ \varphi\ ()\ \textbf{in}\ e'}$$

Let X = locals(e)

From IH, we have $\forall Z \supseteq locals(e)$.
$[\![\Gamma, x : T]\!]$, *o:dyn,* $\Gamma_Z \vdash X.e'\{Post(Z\backslash X)\ t'\}$

Rewriting IH ***as*** *a triple* ***and*** *rearranging refinements for* $[\![x : t]\!]$

(1) $[\![\Gamma]\!]$*,o:dyn,*$\Gamma_{Z,x:t} \vdash \{Pre\}\ e'\ \{Post(Z\backslash X)\ t'\}$
where *Pre h0 = LocalsOK X h0* $\wedge$
$(\exists\ h.\ Witness\ h \wedge Initialized\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h)$

We note that:

$\{\ \lambda\ h.Initiated\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\ \}$
witness()
$\{\lambda\ \_\ \_\ \_ \exists h\ .\ Witness\ h \wedge Initiated\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\}$

and

$\{\lambda\ \_ \exists h\ .\ Witness\ h \wedge Initiated\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\}$
mkLocalInv $\varphi\ X$
$\{\lambda\ \_\ \_\ h0.\ LocalsOK\ X\ h0 \wedge \exists h\ .\ Witness\ h \wedge Initiated\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\}$

Thus, we derive:

(2) $[\![\Gamma]\!]$*; o:dyn;* $\Gamma_{Z,x:t} \vdash$
$\{\lambda\ h.Initialized\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\ \}$
witness(); mkLocalInv $\varphi\ X$
$\{\lambda\ \_\ \_\ \_ Pre\}$

Sequencing (2) ***and*** *(1),*

(3) $[\![\Gamma]\!]$*; o:dyn;* $\Gamma_{Z,x:t} \vdash$
$\{\lambda\ h.Initialized\ h[x]\ h \wedge \psi_t\ h[x][\texttt{"0"}]\ h\ \}$
witness(); mkLocalInv $\varphi\ X$*;* e' *//writing is* ***as*** *body below*
$\{Post\ (Z\backslash X)\ t'\}$

Writing (3) ***in*** *iDST monad:*

$[\![\Gamma]\!]$*,o:dyn,*$\Gamma_{Z,x:t} \vdash$ *body : iDST dyn (****fun*** *post h0.*
Initialized h0[x] h0 $\wedge\ \psi_t\ h0[x][\texttt{"0"}]\ h0 \wedge$
$\forall v, h1.\ Post(Z\backslash X)\ t'\ h0\ v\ h1 \Longrightarrow P\ v\ h1)$

rewriting, weakening, ***and*** *permuting:*

$[\![\Gamma]\!]$*,*Γ_Z*,o:dyn,this:dyn,x:dyn*$\{TypeOf\ x = ORef\ \varphi\ Inv\} \vdash$ *body : iDST dyn*
((WP $\psi_t\ \psi'_t$*) o x this)*

Using TAbs ***in*** *three times, mkFunAbs call is* ***type*** *checked. The return value* ***of*** *mkFunAbs* ***with*** *RTTI tagging gives us the required postcondition*

Case T-D*: Similar to TRec.*

Case T-Sub*: Follows directly from Lemma 3.*

Case A-SetTag,A-Wrap*: Follow directly from the specifications.* □

```
val allocInv: 'P::(obj ⇒ E)
  →caller:dyn
  →flds:obj
  →iDST dyn (fun 'Post h ⇒
      Admissible 'P
      ∧ not (Initialized flds)
      ∧ (∀ l p. ¬InHeap h (TLref l) ∧ AsE l=PT 'P Inv
          ⟹ 'Post (Obj l)
               (UpdHeap h (TLref l) (ConsObj flds "@proto" p))))

val setLocal: caller:dyn
  →l:dyn
  →v:dyn
  →iDST unit (fun ('Post::result dyn ⇒ heap ⇒ E) h ⇒
      (TypeOf l=ORef φ Inv
       ∧ not(Initialized h[l])
       ∧ Initialized v
       let flds = Init({"0"↦v}) in
       ∧ (∀ p. 'P (flds["@proto"] ← p))
       let h' = (h[l] ← (flds["@proto"] ← h[l]["@proto"])) in
       ∧ (Witness h' ⟹ 'Post (V ()) h')))
let setLocal 'P caller l v =
    setInv 'P caller l {Init({"0"↦v})}; witness()

val readVar: caller:dyn
  →l:dyn
  →iDST unit (fun ('Post::result dyn ⇒ heap ⇒ E) h ⇒
      (TypeOf l=ORef φ Inv
       ∧ (∃ h'. Witness h' ∧ Initialized h'[l])
       ∧ ((∀ h'. Witness h' ⟹ δ h' h)
            ⟹ 'Post (V h[l]["0"]) h))
let readVar caller l = recall();selectInv φ f x "0"

val setInv: 'P::(obj ⇒ E)
  →caller:dyn
  →l:dyn
  →flds:obj
  →iDST dyn (fun ('Post::result dyn ⇒ heap ⇒ E) h ⇒
      (TypeOf l=ORef 'P Inv
       ∧ not(Initialized h[l])
       ∧ InitFields flds
       ∧ Initialized flds
       ∧ (∀ p. 'P (flds["@proto"] ← p))
       ∧ 'Post (V Undef)
            (h[l] ← (flds["@proto"] ← h[l]["@proto"])))))

  →fldsf:fn
  →v:dyn
  →iDST dyn (fun 'Post h ⇒
      TypeOf l = ORef 'P Inv
      ∧ not (Initialized h[l])
      ∧ f ∈ dom h[l]
      ∧ ∀o. o=Init (ConsObj (SelHeapObj h l) f (Data v))
        ⟹ (ReachableOK o h
           ∧ 'P o
           ∧ 'Post (V Undef) (UpdHeap h (TLref (GetLoc l))
                                (TO 'P Inv o))))
```

```
val applyT: targ:typ → tret:typ
  → caller:dyn
  → callee:dyn{callee.tag=Abs}
  → this:dyn
  → args:dyn{TypeOf args=ORef φ Inv}
  → iDST dyn (fun 'Post h ⇒
        ψ_{targ→tret} callee h
        ∧ Initialized h[args]
        ∧ ψ_targ h[args]["0"] h
        ∧ ∀x h'. (ResultIs x (λ d. ψ_tret d h')
                    ∧ ∀Z.LocalsOK Z h ⟹ LocalsOK Z h')
                    ⟹ 'Post x h')
val applyAny: targ:typ → tret:typ
  → caller:dyn
  → callee:dyn{callee.tag=Abs}
  → this:dyn
  → args:dyn{TypeOf args=ORef φ Inv}
  → iDST dyn (fun 'Post h ⇒
      TypeOf callee=WP ψ_targ ψ_tret
      ∧ Initialized h[args]
      ∧ ψ_targ h[args]["0"] h
      ∧ ∀x h'. (ResultIs x (λ d. ψ_tret d h')
                  ∧ ∀Z.LocalsOK Z h ⟹ LocalsOK Z h')
                  ⟹ 'Post x h')


val mkFunAbs : 'Tx::(dyn⇒ dyn⇒ dyn⇒ (result dyn⇒ heap⇒ E)⇒ heap⇒ E)
  → s:string
  → (o:dyn
      → iDST (this:dyn → args:dyn → iDST dyn ('Tx o args this))
               (fun 'Post h ⇒ ∀(x:dfun ('Tx o)). 'Post (V x) h))
  → iDST dyn (fun 'Post h ⇒
      (∀ o code x h' flds.
         (not (o.loc.ref ∉ dom h)
          ∧ o.tag=Abs
          ∧ (h' = (UpdHeap h o.loc.ref flds))
          ∧ IsFreshFunObj o h'
          ∧ (x=Fun 'Tx o code)
          ⟹ 'Post (V x) h')))

let mkLocalInv caller φ () = allocInv caller φ [("0",Num 0)]
let tagEq 'P caller l f s = selectInv 'P caller l f = Str s
type bindTX 'a 'b 'Tx1 'Tx2 = fun 'Post ⇒
      ('Tx1 (fun (x:result 'a) (h:heap) ⇒
          (∀ (e:exn). (x=E e) ⟹ 'Post (E e) h) ∧ (* exception *)
          ((x=Err) ⟹ 'Post Err h) ∧ (* error *)
          (∀ (v:'a). (x=V v) ⟹ 'Tx2 v 'Post h))) (* normal *)
type returnTX 'a (x:'a) = fun 'Post ⇒ 'Post (V x)
```

Figure 7. Signatures of functions used in the translation. For the authoritative verified implementation, please see `JSVerify.fst` on the web. We use a slightly more imprecise spec here for *applyAbs* than in our verified implementation, where we give a full spec. The imprecise spec is sufficient for our metatheory, although the full spec is useful in verifying clients of *JSVerify*. We also give a slightly more general specification of *mkFunAbs* than in our verified implementation. This is to enable stating properties on the function object at allocation time.