Towards JavaScript Verification with the Dijkstra State Monad

Nikhil Swamy\(^1\) Joel Weinberger\(^2\) Cole Schlesinger\(^3\) Juan Chen\(^1\) Benjamin Livshits\(^1\)
Microsoft Research\(^1\) UC Berkeley\(^2\) Princeton University\(^3\)

Abstract

Several special-purpose systems have been proposed to analyze programs in JavaScript and other dynamically typed languages. However, none of these prior systems support automated, modular verification for both higher-order and stateful features.

This paper proposes a new refinement of the state monad, the Dijkstra state monad, as a way of structuring specifications for higher-order, stateful programs. Relying on a type inference algorithm for the Dijkstra monad, we obtain higher-order verification conditions (VCs) for programs that use a dynamically typed higher-order store. Via a novel encoding, we show that these higher-order VCs can be discharged by an off-the-shelf automated SMT solver.

We put the Dijkstra monad to use by building a tool chain to verify JavaScript programs. Our tool chain begins by translating JavaScript programs to FS, a dependently typed dialect of ML. Within FS, we define a library for dynamic typing idioms based on the Dijkstra monad. We then infer and solve precise verification conditions for translated JavaScript clients of this library.

We report on our experience using this tool chain to verify a collection of web-browser based dependent type checker to generate and prove VCs. We have used our tool chain to verify a collection of JavaScript web-browser programs to FS⋆, a subset of FS, once in F⋆, we define a library for dynamic typing idioms based on the Dijkstra monad. We then infer and solve precise verification conditions for translated JavaScript clients of this library.

We report on our experience using this tool chain to verify a collection of web-browser based dependent type checker to generate and prove VCs. We have used our tool chain to verify a collection of JavaScript web-browser programs to FS⋆, a subset of FS, once in F⋆, we define a library for dynamic typing idioms based on the Dijkstra monad. We then infer and solve precise verification conditions for translated JavaScript clients of this library.

1. Introduction

For some years now, JavaScript has been the language of choice for client-side web programming. Recent trends indicate that it is also poised to be used widely outside the confines of a web browser, e.g., for Metro applications in Windows 8. A chief characteristic of the language is its highly dynamic nature, which can make it easy to use at least for small programs. However, the dynamism also makes program verification for JavaScript a significant challenge.

1.1 Challenges with verifying JavaScript programs

A first challenge in verifying JavaScript programs is identifying its semantics. JavaScript involves many subtle features, including prototype hierarchies, unusual scoping rules, implicit parameters and conversions. Guha et al. (2010) provide a useful semantics for JavaScript by showing how to desugar JavaScript into a simpler, dynamically typed lambda calculus called JS. Closely following JS, we give a semantics to JavaScript by translating it to JS⋆, a subset of a dialect of ML called F⋆ (Swamy et al. 2011).

To illustrate some key features of JavaScript’s semantics, we present the programs in Figure 1. The JavaScript program at the top is translated to the JS⋆ program shown below. The JS⋆ program is in the ML-like syntax of F⋆, and is typed against an F⋆ library JSPrim that defines a type dyn for dynamically typed values, with constructors like Str, Int, and Fun to inject values into the dyn type. JSPrims also provides functions like allocObject, select, and update to create heap-allocated objects and to read and modify their fields.

```javascript
1 function foo(x) { this.g = x.f + 1; }
2 foo((f:0));
3 foo = 17;
```

```javascript
1 let foo this args = let x = select args "0" in
2 update this "g" (plus (select x "f") (Int 1)) in
3 update global "foo" (Fun foo);
4 let args = let x = update (allocObject()) "f" (Int 0) in
5 update (allocObject()) "0" x in
6 apply (select global "foo") global args;
7 update global "foo" (Int 17)
```

Figure 1. A JavaScript program (top) and its translation to JS⋆.

The JavaScript function foo is translated to the let-bound λ-term foo in JS⋆, with two arguments: this, corresponding to the implicit this parameter in JavaScript, and an argument args, an object containing all the (variable number of) arguments that a JavaScript function may receive. Just like in JavaScript itself, objects in JS⋆ are dictionaries indexed by string-typed keys, rather than containing statically known field names. In the body of foo, the x argument corresponds to the value stored at the key "0" in the args object.

At line 3 of the JS⋆ program, the function foo is stored in the global object (an implicit object in JavaScript) at the key "foo". At line 6 we see that a function call proceeds by reading a value out of the global object, calling the apply library function, which checks that its first argument is a Fun f, and then applies f to its next two arguments—here, the global object, the receiver object for the call; and the args object containing a single argument. At line 7, we update the global object storing the integer 17 at the key "foo".

Now, suppose that one wished to prove that the call to the addition function plus at line 2 always returned an integer. One must prove that all callers of foo pass in an object x as the zeroth value in the args object and that x has a field called "f", which contains an integer value. But, even discovering all call-sites of foo is hard—all (non-primitive) function calls occur via lookups into an untyped higher order store (line 6), and this store is subject to strong updates that can change the type of a field at any point (line 7). The problem is similar to verifying C programs that call functions via void* pointers stored in the heap—in JavaScript, nearly all function calls are done in this style.

1.2 Our contributions

Figure 2 depicts the verification methodology and tool chain developed by this paper. As already mentioned, we begin by translating JavaScript to JS⋆, a subset of F⋆. Once in F⋆, we rely on its SMT-based dependent type checker to generate and prove VCs. We have used our tool chain to verify a collection of JavaScript web-browser extensions for the absence of runtime errors. Aside from annotating loop invariants, verification was automatic.

Our main technical contributions provide a new way of structuring specifications for higher-order, stateful programs, and of in-
ferring and automatically solving VCs for these programs as part of
an end-to-end verification system for JavaScript. Specifically:
— We provide a compiler, JS2JS⋆, to translate JavaScript programs
to JS⋆. Our work adapts λJS, retargeting it to a typed language.
— We present the Dijkstra state monad, a new variant of the Hoare
state monad of Nanevski et al. (2008). The Dijkstra monad equips
a state monad with a predicate transformer (Dijkstra 1975) that
can be used to compute a pre-condition for a computation, for
any context in which that computation may be used. (§3)
— We present JSPrims, a library of dynamic typing primitives in F⋆.
This library includes the definition of a type dyn, a new refinement
of type dynamic (Cartwright and Fagan 1991; Henglein 1994). We
make use of the Dijkstra monad within our type dyn to refine the
type of functions: the refinement of a function is the predicate trans-
former corresponding to its weakest pre-condition. We verify the
implementation of JSPrims against a strong specification, ensuring
assertion safety for well-typed clients of JSPrims. (§4)
— We provide a type inference algorithm that allows stateful pro-
grams written in a direct style to be interpreted in the Dijkstra state
monad, yielding a VC for the program. We prove our inference al-
gorithm sound and complete. Informally, this means that we
can compute a VC for any loop-free JS⋆ program, and if that VC is
provable, then the JS⋆ program has no failing assertions. As usual,
programs with loops require loop invariant annotations. (§5)
— The proof obligations produced by our inference algorithm make
heavy use of higher-order logic. Our fifth contribution is a new
 technique that incrementally compiles higher-order verification
conditions computed for JS⋆ programs to a set of first-order proof
obligations that can be automatically discharged by an off-the-shelf
SMT solver—we use Z3 (de Moura and Bjørner 2008). (§6)
— We extend JSPrims to include a partial specification of common
APIs used by JavaScript programs, including a fragment of the
Document Object Model (DOM). Our specification includes prop-
erties of recursive data structures (like DOM elements) using induc-
tive predicates, and giving specifications to higher-order functions
(including in some cases, functions of the third order). (§7)
— We evaluate our work experimentally by collecting a library of JavaScript web-browser extensions (on the order of 100 lines
each) for various properties, including the absence of JavaScript
web-browser extensions on the order of 100 lines each for various properties, including the absence of JavaScript runtime errors. In each case, apart from carefully specifying loop invariants, verification was automatic. Our tool chain makes use of
Gatekeeper (Guarnieri and Livshits 2009), an efficient but generally
unsound pointer analysis for JavaScript programs. We show how to
soundly integrate such a tool with our verification methodology
to improve the efficiency of verification. (§7)
As such, ours is the first tool to enable precise, modular, verifica-
tion for a sizeable subset of JavaScript, including its higher-order
and stateful features. In contrast, prior work on analysis for dynami-
cally typed programming languages either rely on a whole-program
abstract interpretation (Jensen et al. 2009), or provide a modu-
lar type-based analysis for languages that lack higher-orderliness
or state, or both (Bierman et al. 2010; Chugh et al. 2012; Tobin-
Hochstadt and Felleisen 2010). Furthermore, our approach conven-
iently rides on the mechanized soundness and certified implementa-
tion of F⋆, which we reuse with little modification.

Limitations. The main limitations of this paper are with regard to
the subset of JavaScript we discuss. We focus primarily on the
difficulty of modeling and reasoning about JavaScript’s higher-
order store. Our presentation elides the following main features of
JavaScript: exceptions, control operators like breaking to a label,
treating functions as objects, and prototype chain traversals. These
omissions are not fundamental limitations; a concurrent submission
on secure compilation to JavaScript builds on the ideas of this paper
provide a high-fidelity model of JavaScript using the Dijkstra
monad within F⋆, including all of the missing features mentioned
above as well as several others. However, neither paper provides a
complete model of the DOM. This paper models some elements of
DOM structure, like parent/child relationships, but does not
precisely model the (anti-)aliasing patterns of the DOM, and does
not model asynchrony. Finally, this paper ignores eval. Although
not suitable for verification, the other paper models the power of an
eval-capable attacker.

Supplementary materials associated with this paper are available
from http://research.microsoft.com/fstar. This includes a compiler
download, a more general formulation of inference for
the Dijkstra monad, proofs of all the theorems in this paper, and the
complete JSPrims library.

2. A brief review of F⋆

F⋆ is a variant of ML with a similar syntax and dynamic semantics
but with a more expressive type system based on dependent types.
In this paper, aside from the types of ML, we limit ourselves to a
few main F⋆-specific typing constructs.

The type system of F⋆ is partitioned into several subsystems
using a discipline of kinds. We use two base kinds in this paper.
The kind ∗ is the kind of types given to computations. In contrast,
the kind E is the kind of types that are purely specification.
A sub-kinding relation places the kinds in the order ∗ ≤ E.

Ghost refinement types of the form kxt\{ϕ\} represent values kvt
for which the formula ϕ[k/v/x] is derivable from the hypothesis in
the current typing context. The formula ϕ is itself a type of kind
E—it is purely specification and has no runtime representation.
Thus, kxt\{ϕ\} is a subtype of kxt. Dependent function arrows kxt→kt′
represent functions whose domain is named x of type t and whose
code-domain is of type t′, where t′ may depend on x. For non-
dependent function types, we simply write t→t′. We write kxt→kt′
for a dependent pair, where x names the first component and is
bound in t′. We also use function arrows whose domain are types
a of kind k and whose code-domain are types t, written a:k→t, i.e.,
these are types polymorphic in types a of kind k.
Aside from the base kinds (⋆ and E), we use product kinds ‘a:k ⇒ k’ and ‘x:t ⇒ t’ for type constructors or functions. For example, the list type constructor has kind ⋆ ⇒ ⋆, while the predicate Neq (for integer inequality) has kind int ⇒ int ⇒ E. We also write functions from types to types as fun ‘(‘a:k) ⇒ t’, and from terms to types as fun ‘(‘x:t) ⇒ t’. For example, fun ‘(‘x:int) ⇒ Neq ‘x 0’ is a predicate asserting that its integer argument ‘x’ is non-zero. As a general rule, the kind of a type is ⋆, if not explicitly stated otherwise.

F* is also parametric in the logic used to describe program properties in refinement formulas φ. These formulas φ are themselves types (of kind E, and so purely specification), but these types can be interpreted in a logic of one’s choosing. In this paper, we use a higher-order logic, extended with a theory of equality over terms and types, a theory of linear integer arithmetic, datatypes, and a select/update theory of functional arrays (McCarthy 1962) (useful for modeling heaps). Except for the higher-order constructs (which we handle specially), all decision problems for the refinement logic are handled by Z3, the SMT solver integrated with F*’s typechecker. The binary predicates EqVal and EqTyp below are provided by F*’s standard prelude and represent syntactic equality on values and types respectively. For brevity, we write v=v’ and t=t’ instead of EqVal v v’ and EqTyp t t’, respectively.

type EqVal ‘(‘a:⋆) :: ‘a ⇒ ‘a ⇒ E

type EqTyp :: E ⇒ E ⇒ E

type Of ‘(‘a:⋆) :: ‘a ⇒ E

Every type and data constructor in an F* program is, by default, treated as an injective function in the logic (i.e., according to the theory of datatypes). To override this default behavior, we tag certain declarations with various qualifiers. For example, the declaration of TypeOf above is tagged logic tfun, indicating that it is to be interpreted as a non-injective type function from values to E-types in the refinement logic.

Values can also be provided with interpretations in the logic. For example, to model a map type as a functional array in Z3, we write the following declaration in F*:

logic array sel : heap ⇒ ref ‘a ⇒ ‘a
logic val sel : heap ⇒ ref ‘a ⇒ ‘a
logic val upd : heap ⇒ ref ‘a ⇒ ‘a → heap
logic tfun InDom : heap ⇒ ref ‘a ⇒ E

This provides an abstract type called heap, with four interpreted functions in the logic: sel selects the value from the heap at the specified reference; upd updates the heap at the specified reference location; emp is the empty heap; and InDom is an interpreted predicate to test whether a reference is allocated in the heap. The logic val tag records that these functions are interpreted in the logic only— they may be used only within specifications.

The system and metatheory of F* has been formalized in Coq, showing substitutivity and subject reduction, while a separate progress proof has been done manually. These type soundness results (modulo the soundness of the refinement logic) imply the absence of failing assertions in any run of a well-typed F* program.

The JS* language. JS* is a small subset of F*—its syntax is shown below. Values v include variables, n-ary lambdas, and n-ary data constructor applications. Formally, we encode other common constants (strings, integers, unit, etc.) as nullary data constructors, although our examples use the usual constants. Expressions include values, function application (as in F*, we require the arguments to be values), let bindings, and pattern matching.

Syntax of JS*

\[
\begin{align*}
  v & ::= \text{ x } \mid \lambda x.e \mid C.v \\
  e & ::= v \mid v \cdot f \mid \text{ let } x = e_1 \text{ in } e_2 \mid \text{ match } v \text{ with } C.f \rightarrow e_1 \text{ else } e_2
\end{align*}
\]

No JavaScript-specific constructs appear in JS* directly. Instead, we provide an F* library, JSPrims, which is used by every JS* program produced by the JS2JS* translation. For example, the data constructors for type dyn are also defined by JSPrims. Additionally, as shown in Figure 1, functions to manipulate state are provided by the JSPrims library. We give specifications to these primitive functions (and hence also to their JS* clients) using the Dijkstra monad, which we discuss next.

3. The Dijkstra state monad

Monads provide a convenient way of structuring effectful computations in functional programs. For example, the type ST ‘a can be given to an expression of type heap → ‘(‘a x heap), which is the type of a computation transforming an input heap to a result of type ‘a and an output heap. To give specifications to such stateful computations, Nanevski et al. propose the Hoare monad, refining the ST monad with predicates corresponding to a pre-condition on the input heap and a post-condition on the result and output heap, e.g., h:heap [pre h] => (x:h:heap [post x h]).

Inferring verification conditions for the Hoare monad poses two difficulties. First, weakest pre-condition calculi for the Hoare monad generally rely on a programmer-supplied post-condition, to compute a pre-condition by “pushing the post-condition” backward. However, for JS*, no such post-condition may exist. Second, when composing computations in the Hoare monad, one needs a subsumption rule to weaken pre-conditions and strengthen post-conditions. This is difficult to use in a type inference algorithm, since unification must now be performed modulo subtyping.

We propose to refine the state monad with a predicate transformer, a function that computes the weakest pre-condition for the computation from an arbitrary post-condition. We call the resulting monad the Dijkstra monad, and show its definition in F* below.

Definition of the Dijkstra state monad in F*

\[
\begin{align*}
  \text{type } & \text{ DSTM ‘(‘a:⋆) = } \phi ::(‘a ⇒ heap ⇒ E) ⇒ heap ⇒ E \\
  & \psi ::(‘a ⇒ heap ⇒ E) ⇒ h:heap [φ ψ h] ⇒ (x:‘a ⇒ (h:heap [ψ x + h])) \\
  \text{val return} :: ‘a → ‘a ⇒ E \Rightarrow ‘a → heap \Rightarrow ‘a \Rightarrow heap \Rightarrow E \Rightarrow ‘a \Rightarrow heap \Rightarrow E \\
  \text{let return} :: ‘a x ψ h = x, h \\
  \end{align*}
\]

At line 1, we define the type DSTM ‘a φ as the type of a computation which, for any post-condition ψ, produces a result x, h satisfying ψ x + h, given an input heap h satisfying φ . That is, φ is the predicate transformer computing a pre-condition on the input heap from any post-condition on the result and output heap.

We also define two operations return and bind, to inject pure values into the monad, and to sequentially compose two monadic computations. It is easy to check that bind is associative and that return is both a left- and right-identity for bind, making DSTM monad-like.

The type of return shows that the weakest pre-condition of a value x with respect to some post-condition ψ, is ψ x, since the heap does not change at all. The type for bind allows any two stateful computations to be composed (so long as the second can consume the result of the first). The predicate transformer for the composed computation is, in effect, the composition of the individual predicate transformers—unlike the Hoare monad, we do not need a subsumption rule to compose computations.
The type constructor `constructed simply records the predicate transformer capturing the semantics of its arguments. The type constructor `Function serves simply to coerce the predicate transformer to kind `E.

Figure 3. A refinement of type dynamic

Writing post-condition polymorphic specifications for the DST-monad is also convenient, e.g., one does not have to directly state that return leaves the input heap `h unchanged. One can also easily give specifications to stateful operations in the DST-monad. For example, we show the type of the infix assignment operator for references `(==) below:

```plaintext
val (x := v) : `a → `a → DST unit
  (fun (Post := unit ⇒ heap ⇒ E) ⇒ (h; heap) ⇒ Post () (upd h x v))
```

This says that the weakest pre-condition of `x := v is to prove the post-condition on the initial heap, updated at reference `x with the value `v (expressed using the interpreted function `upd for heap update shown earlier). Without this polymorphic style one would need extra conditions to relate the output heap to the input heap, for example, by making use of two-state predicates as post-conditions and stating that only the reference `x was changed.

4. A library for dynamic typing in `F*

We now describe an `F* library, `JSPrims, which defines various constructs suitable for verifying `JS* programs. There are two key innovations. First, we make use of the Dijkstra monad to define a type-the type `stmt objects that—serve as isomorphisms to the natural numbers, the type of heap locations. The `Unref constructor is for the undefined value in `JS*.

The case for `Fun merits closer attention. As we will see, the JS2JS* translation translates every JavaScript function to a binary, curried function. The first argument is for the implicit `this parameter (made explicit in the translation), and the second parameter is for all the other arguments represented as a dictionary. To a first approximation, the type of `Fun is `((dyne → dyn → dyn) → dyn), but this is, of course, too imprecise. We need a more precise type that can capture the behavior of the function, including its effects on the heap—the predicate transformers of the DST monad come to mind. We interpret every `JS* function in the DST monad, and give them types of the form `thisdyn → args dyn → DST dyn (`Tx args this), for some predicate transformer `T. Then, the refinement on the `Fun constructor simply records the predicate transformer capturing the semantics of its arguments. The type constructor `Function serves simply to coerce the predicate transformer to kind `E.

1. `type heapcell = Cell of `a: `E → `d: dyn `{TypeOf d = `a} → heapcell
2. `type fn = string `{* field names *}
3. `{* Map from field names to heapcells *}
4. `logic array(SelCell, UpdCell, EmpF, InFields) `type fields
5. `logic selCell: fields → fn → heapcell `{* select a single field *}
6. `logic val UpdCell: fields → fn → heapcell `{* upd a single field *}
7. `logic val EmpF: fields `{* empty fields *}
8. `logic fun type InFields: fields → fn = `E `{* fn in domain of fields *}
9. `{* Map from object locations to fields *}
10. `logic array(SelObj, UpdObj, EmpInDom) `type heap
11. `logic val SelObj: `heap → `loc → fields `{* select an object *}
12. `logic val UpdObj: `heap → `loc → fields → heap `{* update an object *}
13. `logic val Emp: `heap `{* empty heap *}
14. `logic fun type InDom: `heap ⇒ `loc ⇒ `E `{* loc in domain of heaps *}
15. `{* Derived forms *}
16. `logic val Loc: `dyn ⇒ `loc `{* projecting a loc from an obj *}
17. `assume `Ev: (Loc (Obj `l) = `l)
18. `logic val Select: `heap ⇒ `dyn ⇒ fn ⇒ dyn `{* select a field *}
19. `assume `h l f v: (`Select `h l f = `v)
20. `⇐⇒ (`SelCell (SelObj `h (Loc `l)) = (`Cell `v))
21. `logic fun type SelectT `type fields
22. `assume `h l f `a: (`SelectT `h l f = `a)
23. `⇐⇒ (`SelCell (SelObj `h (Loc `l)) = (`Cell `a `a))
24. `logic val Update: `heap ⇒ `dyn ⇒ `fn ⇒ dyn `{* update a field *}
25. `assume `h l f v: (`UpdCell (SelObj `h l f v) = `Cell) `TypeOf v `v)
26. `type HasField: `heap ⇒ `dyn ⇒ `E
27. `assume `h o f: `HasField `h o f
28. `⇐⇒ (`InDom `h (Loc `l o) & & `InFields (SelObj `h (Loc `l o)) `f)
29. `{* Private API of `JSPrims *}
30. `private val selcell: `h: `heap ⇒ `loc ⇒ `fn
31. `⇐⇒ (`option heapcell `{HasField `h (Obj `l) `f})
32. `⇐⇒ (`oSome (SelCell (SelObj `h `l `f))
33. `private val updcell: `h: `heap ⇒ `loc ⇒ `fn ⇒ `v: `dyn
34. `⇐⇒ (`h: `heap `{hUpd `h `l `f `v}
35. `private val alloc: `unit ⇒ `heap
36. `⇐⇒ (`(x dyn = `h: `heap `{not (InDom `h (Loc `x))})
37. `& & `h = (`UpdObj `h (Loc `x) `EmpF))

Figure 4. A logical model of the `JS* heap and its private API

Our full library includes cases for the other JavaScript primitive types as well. Note, from here on, unless we think they specifically aid understanding, we omit kind annotations on type variables for conciseness and clarity.

4.2 Modeling the JavaScript heap

We have two main design considerations in modeling the JavaScript heap. First, to model a dynamically-typed higher order store subject to strong updates, we model each heap cell as a pair of a `dyn value and its associated type refinement. An update to a heap cell updates both the value and the associated type. Second, while it is possible to model the heap entirely within `F* using association lists from locations and field names to heap cells, this method is poorly suited to automated reasoning. Instead, we rely on Z3’s efficient theories of functional arrays to model the heap.

Figure 4 shows our internal model of the heap. `JS* programs that are clients of this library cannot directly manipulate any of the programmatic elements of this model (e.g., directly call the function `select to access a heap cell). In the next subsection, we present a monadic public API exposed to client programs.

Our model starts with the definition of `heapcell; in effect, an existential package of a type `a and `dyn value `d such that `TypeOf `d = `a. The whole heap is essentially a two-dimensional array of `heapcell, indexed first by object locations and then field names. We start by defining an abstract type `fields, which is interpreted in the logic a map from field names (`fn to `heapcells. The logic functions `SelCell, `UpdCell, `EmpF and the predicate `InFields provide interpre-
tations of this type in the logic as a functional array. Similarly, the type heap is abstract, and is interpreted in the logic as a map from object locations to fields—again with three interpreted functions and one interpreted predicate.

Based on these interpreted array types, we define four derived forms that allow us to manipulate the heap as a two dimensional map indexed by dyn values and field names. The function Loc (line 16) is defined to project the location out of an Obj value. At line 18, the derived form Select is defined to select just the value part of a heap cell, while SelectT at line 21 selects just the type part. We define Update at line 24, and interpret it in the logic as an update to a single field of an object, and, finally, we define a predicate HasField to check if an object contains a field.

Recall that all these logic values are purely specifical—
they may only be used in refinement formulas not in the executable part of the program text. However, real JS' programs must be able to actually read from and write to heap cells. For this, we provide three function symbols, selcell, updcell, and alloc, whose implementation can be in native code, and assert that the specification of these primitive operations corresponds to selecting, updating and allocating objects. Since these functions will be called from our library, we give them refined types to restrict their usage. For example, both selcell and updcell insist that they be called only with locations that are guaranteed to be in the heap.

Our specification for alloc is worth discussing more. It says that alloc returns a heap h' that differs from the initial heap h only in a new location x, where the content of x is initialized to the empty object. In real JavaScript, as in our full library, objects are allocated with a prototype property. As mentioned in §1, for simplicity and space constraints, we elide prototype chains in our JS* model. The public API provides a monadic wrapper allocObject for alloc.

4.3 A Dijkstra monadic public API for JSPrims

We now discuss selected functions in the public API of JSPrims that allow safely reading and writing fields in the heap, and safely applying dynamically typed functions. In designing this API we had to balance two concerns. On the one hand, we want the API specification to be strong enough so that well-typed JS* clients are sure to not have any runtime errors, e.g., they do not attempt to project a field out of an integer value, or apply a non-function. Simultaneously, we want our specification to be weak enough that typical, well-behaved JS* programs can be successfully typechecked.

To ensure that we correctly balance these concerns, we carry out two tasks. First, we provide verified implementations (in F*) of each of the functions in our public API. Thus, from the soundness of F* we have a proof that well-typed clients of JSPrims indeed do not have the assertion failures we forbid. Secondly, while a formal proof demonstrating that our specification is the weakest would be desirable, for the moment, we simply validate empirically that our specification is sufficiently weak, i.e., we show that several typical programs can be checked against this API.

Safe field selection. Informally, for o:dyn and f:fn, the term select o f corresponds to the JavaScript field lookup operation o[f]. More precisely, the specification of select o f below (lines 1–4) shows that it is a computation which produces a dyn value, and satisfies any post-condition “Post when run in an input heap h satisfying the pre-condition at lines 2–3 (“⇒” stands for implication). The caller must prove three properties. The first requires that the type of o be object, since projecting a field from other values is a JavaScript error. The second ensures that field o[f] exists. If the field does not exist, JavaScript semantics permits returning the undefined value, which is, strictly speaking, not an error condition. However, checking for the absence of unexpected undefined values in a JavaScript program is generally considered a good idea (Crockford 2008), so we check for it here. Finally, the client is required to prove the predicate ”Post (Select h o f h) (indicating that select indeed returns the contents of the o[f] field and leaves the heap unchanged), under the assumption that the returned value (Select h o f) has the type associated with it in its heapcell.

Signature and implementation of field selection

```
1 type SelTX o f 'Post h =
2  (TypeOf o)object && HasField h o f &&
3  (TypeOf (Select h o f))= (Post (Select h o f) h))
4 val select: o:dyn → f:fn → DST dyn (SelTX o f)
5
6 let select o f 'Post h = match o with
7   | Undef | Str _ | Int _ | Fun → assert False
8   | Obj l → (match selcell h l f with Some (Cell _ v) → v,h
9   | None → assert False)
```

We also show the implementation of select, to be typechecked against its specification. F* types each pattern branch under the assumption that the scrutiny o is equal to the pattern. Thus, the first branch is typed under the assumption that o = Undef, or o=Str _ , etc. Given our pre-condition that TypeOf o)object, we reach a contradiction, i.e., we are able to prove False. In the last case, we have the assumption that o=Obj _ and from the post-condition of selcell we have that the scrutiny in the nested match construct is equal to SelCell h o f. From our pre-condition HasField h o f, we can prove that the scrutiny is not equal to None, and thus, only the first case of the nested match is reachable. In this case, we return the selected value v and the unchanged heap h. The whole function is verified automatically according to its specification.

Safe field update. Informally, for o:dyn, fn:fn, v:dyn, the term update o f v corresponds to the JavaScript field assignment o[f] = v. The specification of update has a similar form to that of select, and is shown in the listing below. The caller is required, first, to prove that o is an object. Second, o must be in the domain of the current heap, although it may not have the field f yet—JavaScript permits adding fields to an object “on the fly”. Finally, the caller has to prove the post-condition assuming that update o f v returns Undef and that the heap is updated appropriately. The implementation of update, also shown below, is straightforward.

Signature and implementation of field update

```
1 type UpdTX o f v 'Post h =
2  TypeOf o)object && InDom h (Loc o) && 'Post Unfun (Update h o f v)
3 val update: o:dyn → fn:fn → v:dyn → DST dyn (UpdTX o f v)
4
5 let update o f v 'Post h = match o with
6   | Undef | Str _ | Int _ | Fun → assert False
7   | Obj l → (update (match updcell h l f with Some (Cell _ v) → v,h
8     | None → assert False)
```

Safe function application. Informally, for f:fn, this:fn, args: dyn, the term apply f this args corresponds (roughly) to the JavaScript construct this.f(args). As in the other cases, our goal is to ensure that function implementations in JS* (and hence in JavaScript) do not cause errors. There are two things that could potentially go wrong. First, f may not be a function—it is an error in JavaScript to apply, say, an integer. Second, f’s pre-condition may not be satisfied. Addressing both these concerns, we show the type and implementation of apply below.

Signature and implementation of function application

```
1 logic type Unfun :: E ⇒ dyn → dyn → (dyn⇒ heap ⇒ E)⇒ heap ⇒ E
2 assume ∀T.x.T ⇓ Unfun (Function T)=: T
3 assume ∀a.a << (Function a)=: (Unfun a)=Unfun (fun → False)
4 val apply: f:fn → this:fn → args: dyn
5  ⇒ DST dyn (Unfun (TypeOf f) args this)
6 let apply f this args 'Post h = match f with
7   | Undef | Str _ | Int _ | Obj → assert False
8   | Fun T.fn → fn this args 'Post h
```
Intuitively, if we can ensure that \((\text{typeof } f) = \text{Function}\) for some transformer in \(\text{TS}\), then we can rule out the first kind of error (i.e., \(f\) is guaranteed to be a function). Then, to enforce \(f\)’s pre-condition we can simply apply the predicate transformer to the arguments, the post-condition, and the heap to compute the pre-condition, i.e., we require \(\text{TS args this} \) Post h.

Stating these requirements in a form amenable to easy proving takes a little work. First, lines 1–3 define a type function Unfun from arbitrary E-types to types that represent predicate transformers. The interpretation of Unfun (the two assumes) shows that when applied to a type of the form Function in \(\text{TS}\), Unfun simply projects out the transformer in \(\text{TS}\). Otherwise, Unfun returns the False predicate. Using Unfun, the specification of apply is easy: the predicate transformer of apply in this argument is just Unfun (Type of f) args this. If \(f\), the value being applied, can be proved to be a function, then the behavior of apply is just as if \(f\) were applied; otherwise the pre-condition of apply in this argument is unsatisfiable.

The implementation of apply is straightforward. When \(f\) is not a function, Unfun (Type of f) gives us the False pre-condition, which suffices to show that the first case is unreachable. In the second case, we have the pre-condition of fn among our hypotheses, so we can again prove the goal. Now, proving these goals directly using Z3 is infeasible—it does not understand \(\text{TS}\), a first-order SMT solver, is infeasible—it does not understand \(\text{TS}\), a first-order SMT solver, is infeasible—it does not understand.

### 5. Monadic type inference for JS*

This section formalizes a type inference algorithm for JS* programs that are typed against F* libraries (like JSPrims) that make use of the \(\text{DST}\) monad. We prove our algorithm sound by showing that JS* programs are easily elaborated to standard F* programs where they can be checked to have the types we infer—soundness follows from the soundness of F*. Additionally, we show that for programs (typed in a context meeting a certain regularity condition), we can always infer a sound type, i.e., type inference is also complete. The supplementary materials include a detailed formulation of the inference algorithm, and its meta-theory. Here, we present a version of the algorithm specialized to the type \(\text{dyn}\), and quote the theorems from the full version of the paper without providing proofs.

#### 5.1 Type inference for the Dijkstra monad

Figure 5 formalizes two judgments. The first judgment, \(\Gamma \vdash \nu : t\), infers a type \(t\) for a JS* value \(\nu\) from a context \(\Gamma\). For expressions, we have \(\Gamma \vdash \epsilon \mapsto \phi\), which infers a predicate transformer \(\phi\) for an expression \(\epsilon\) from a context \(\Gamma\). Informally, from this latter judgment, we conclude that \(\epsilon\) has type \(\text{DST dyn}\). Thus, all expressions are typed monadically and produce dyn-typed results.

**Syntax.** The types we use include \(\sigma\), prenex quantified scheme types—as in the previous section, we elide kind annotations for compactness. Types \(t\) include type constants \(T\) (like \(\text{int}\), \(\text{bool}\), etc.). We also have refinements of type \(\text{dyn}\). Impure functions have types like \(\text{TS} \rightarrow \text{D}\) \(\phi\), where the co-domain \(\text{D}\) \(\phi\) is an abbreviation for \(\text{DST dyn} \phi\). Finally, we have pure functions \(\text{TS} \rightarrow \text{t}\), which, generally, is used only to describe the type of data constructors. In both arrow constructions, the bound names \(x\) are in scope to the right of the arrow. The formulas \(\phi\) used in types are just drawn from \(\text{F*}\)’s refinement logic, as used in the previous section. One point to note: in the formal system, we write \(\lambda x. \phi\) instead of fun \(\phi\) and \(\lambda x. \phi\) instead of fun \(\phi\).

**Value typing.** Value typing makes use of the type instantiation relation \(\sigma \geq t\). We use this relation non-deterministically in the premises of the first two rules. In the full version of the paper, we show how to easily compute instantiations using unification. Typing abstractions is straightforward—we note that all JS* functions have dyn-typed arguments.

**Expression typing.** The first expression typing rule corresponds to a monadic unit, and allows a dyn-typed value \(\nu\) to be injected into the \(\text{DST}\) monad. The rule follows the signature of return shown earlier, except that, if the value being returned has a refined type, then the weakest pre-condition is guarded by this refinement. The rule for function applications is standard for a dependently typed language. Note that we require the function to be impure and the result to have monadic type. Typing a conditional is straightforward—we type each branch and compute a predicate transformer by guarding the pre-conditions of each branch by the branch condition. Finally, the rule for let-binding corresponds to the monadic bind. As in the signature of bind shown earlier, the conclusion composes the predicate transformers of each sub-expression.

**Soundness.** The full paper defines a type-directed elaboration of JS* into F*, based on the structure of the inference rules in Figure 5. These judgments are written \(\Gamma \vdash \nu : t\) and \(\Gamma \vdash e \mapsto \phi\)—type inference for JS*

\[
\begin{align*}
\sigma & := \forall x. t \\
\Gamma & := (x: \sigma | C: \sigma | \Gamma, \phi) \\
\Gamma \vdash e & := \Gamma \vdash \lambda x. \phi \\
\Gamma \vdash e & := \Gamma \vdash t \\
\Gamma \vdash \nu : e & := \text{dyn} \phi \\
\end{align*}
\]

**Theorem 1** (Type inference is sound). For all \(\Gamma, x, t, v, e, \phi, \theta, e'\), if \(\Gamma \vdash e \mapsto \phi \mapsto e'\), then \(\text{JSPrims} \vdash \Gamma, e \mapsto \phi \mapsto e'\). **DST dyn**: \(\phi\); and if \(\Gamma \vdash v : t \mapsto v'\), then \(\text{JSPrims} \vdash \Gamma, e \mapsto \phi \mapsto e' : t\).

**Completeness.** Our completeness result applies to all JS* programs that are typeable in the standard ML typing discipline while making use of an unrefined type dyn—we write this judgment as \(\Gamma \vdash \text{ML} \Gamma \vdash e : \text{ML}\), and call such programs “ML-typeable”. Here, \(\text{ML}\), \(\Gamma\) is the erasure of a JS* typing context \(\Gamma\) to an ML typing context. Trivially, every program in the image of the JS2JS* translation is ML-typeable. We prove that given a well-formed JS* typing context \(\Gamma\), any ML-typeable program \(e\) in \(\Gamma_{\text{ML}}\) is also typeable by our inference algorithm. Intuitively, our well-formedness condition, written \(\vdash \Gamma \text{ ok}\), requires every higher-order function type in \(\Gamma\) to be
maximally general in the predicate transformers of its function-typed arguments. For example, we permit types in Π of the form \( \forall \psi. (x: \text{dyn} \rightarrow D \psi) \rightarrow D \phi \), since this is a higher-order type that is parametric in the predicate transformer \( \psi \) of its argument. Another example of a type meeting this generality condition is the type of the \( \text{Fun} \) constructor from \( \text{JSPrims} \). In contrast, the generality condition forbids types of the form \( (x: \text{dyn} \rightarrow D \Lambda \psi \cdot \Lambda h. \psi \cdot h) \rightarrow D \phi \).

**Theorem 2** (Type inference is complete). For all \( \Gamma, \psi : \text{ML} \), such that \( \Gamma \vdash \psi \cdot t \cdot \text{Dyn} \) then there exists \( t' \) such that \( \Gamma \vdash \psi \cdot t' \). Additionally, for all \( e \) such that \( \psi \cdot \text{Dyn} \vdash e \cdot \text{Dyn} \), there exists \( \phi \) such that \( \Gamma \vdash \psi \cdot e \Rightarrow \phi \).

**Incompleteness with recursion and loops.** As stated above, our completeness result applies only to loop-free \( \text{JS}^\ast \) programs. As defined, \( \text{JS}^\ast \) programs contain no explicit recursion, however, since the store is higher-order, recursion through the store is possible. For programs that are recursive through the store, our type inference algorithm computes a candidate pre-condition, but, without a programmer-supplied annotation, that condition will in general be unprovable. Space constraints preclude a detailed discussion of recursion through the store. More commonly, we allow \( \text{JS}^\ast \) programs to contain loops and provide combinators in \( \text{JSPrims} \) for JavaScript’s for in- and while-loops. We discuss programs that use these combinators in §7. Suffice to say for the moment that calls to these combinators require the programmer to supply additional predicate arguments that stand for loop invariants—clearly, our inference algorithm is incomplete for such loop programs.

### 5.2 Monadic inference in action

We illustrate our type inference algorithm using the \( \text{JS}^\ast \) program in Figure 1. The general syntactic shape of the program is

\[
\text{let } \text{foo} = \text{in } e \quad \text{and we assume that the transformer for } \text{foo} \text{ has already been inferred as } \text{FooTX}.
\]

We show the derivation for the remainder of the program, writing the transformer \( \phi_1 \) for the predicate transformer inferred for the \text{let}-binding at the line just below.

**An example derivation**

1. \[\text{let } \text{foo} \text{ this : } D (\text{FooTX} \text{ args this }) \Rightarrow \ldots \text{ in }\]
2. \[\phi_0 = \Lambda \psi. \Lambda h. (\text{TypeOf } (\text{Fun } \text{foo})) = (\text{Function } \text{FooTX})\]
3. \[\Rightarrow (\text{fun } \text{foo } \Rightarrow \phi_0 \text{; } h) (\text{Fun } \text{foo})\]
4. \[\text{let } \text{fun} = (\text{Fun } \text{foo}) \text{ in }\]
5. \[\phi_1 = \Lambda \psi. \text{UpdateTX} \text{ global } "\text{foo}" \text{ foo } \text{ fun} \text{; } \Rightarrow (\phi_1 \cdot \psi)\]
6. \[\text{let } _\text{u} = \text{update global } "\text{foo}" \text{ fun} \text{ in }\]
7. \[\phi_2 = \Lambda \psi. \text{ArgsTX} \text{ fun } \Rightarrow (\phi_2 \cdot \psi)\]
8. \[\Rightarrow (\text{fun } \text{foo } \Rightarrow \phi_2 \text{; } \psi)\]
9. \[\text{let } _\text{f} = \text{select global } "\text{foo}" \text{ fun} \Rightarrow (\phi_2 \cdot \psi)\]
10. \[\text{let } _\text{f} = \text{select global } "\text{foo}" \text{ fun} \Rightarrow (\phi_2 \cdot \psi)\]
11. \[\phi_3 = \Lambda \psi. \text{Unfun } (\text{TypeOf } f) \text{ args global } (\text{fun } \Rightarrow (\phi_2 \cdot \psi))\]
12. \[\text{let } _\text{u} = \text{apply } f \text{ global args in }\]
13. \[\phi_4 = \Lambda \psi. \text{h. } \text{TypeOf } \text{int } 17 \Rightarrow \text{Int } 17 = \text{int } \Rightarrow \text{UpdTX} \text{ global } "\text{foo}" \text{ (int } 17 \text{ ) } \psi h\]
14. \[\text{let } _\text{f} = \text{int } 17 \text{ in }\]
15. \[\text{update global } "\text{foo}" \text{ i ; } \text{UpdTX} \text{ global } "\text{foo}" \text{ i }\]

It is convenient to start at the bottom-most let-binding, although our inference algorithm proceeds top-down. At the call to \text{update} at line 15, we just apply \text{UpdTX}. To compose this with the transformer of \text{int } 17, we use the rule for returning values into the monad and compose the transformers to obtain the transformer \( \phi_1 \) at line 13. Notice that the pre-condition is guarded by the typing assumption of \text{int } 17. Proceeding upwards, we compose the transformer \( \phi_1 \) with the transformer from the call to \text{apply}, obtaining \( \phi_2 \). Traversing the let-binding at line 10, notice that we close \( \phi_0 \) over the let-bound variable \( e \), and compose it with \text{SetTX} obtaining \( \phi_1 \). We proceed in this manner until we reach \( \phi_0 \) at line 3, which gives the predicate transformer for the entire expression. Notice that for any post-condition \( \psi \), the pre-condition we compute is guarded by the typing assumption of \text{Fun } \text{foo} which records the transformer \text{FooTX}.

To verify the program against a particular post-condition, say, \( \lambda x. \text{True} \), we apply \( \phi_0 \) to this formula obtaining a predicate \( \phi \) on the initial heap. We apply \( \phi \) to a variable \( h_0 \) bound in the initial typing context \( \Gamma_0 \) representing the initial heap. In addition to the definitions and assumptions in the \( \text{JSPrims} \) library (e.g., the definition of the \text{dyn} type and the assumptions about \text{Unfun}, etc.), \( \Gamma_0 \) also includes assumptions about the initial heap, e.g., that it contains the \text{global} object. (In §7, we give a concrete \text{InitialHeap} predicate for JavaScript programs.) Our verification goal then is to show that in \( \Gamma_0 \) the formula \( \phi \cdot h_0 \) is derivable, or, equivalently, that the context \( \Gamma_0 \) extended with the assumption \( \neg (\phi \cdot h_0) \) is unsatisfiable.

### 6. Solving verification conditions

Given an initial context \( \Gamma_0 \), our goal is to prove (automatically) that \( \Gamma_0 \cdot \neg (\phi \cdot h_0) \) is unsatisfiable. This problem is well-understood when \( \Gamma_0 \) and \( \phi \cdot h_0 \) are first-order formulas. A range of program verification tools, including F*, rely on efficient SMT solvers which have heuristic support for reasoning about quantified formulas to successfully discharge first-order proof obligations. However, the proof obligations produced for \( \text{JS}^\ast \) programs are not always first-order formulas. While some experimental automated solvers for certain higher-order logics are being developed, these remain inefficient, and provide little support for reasoning about the theories we need for \( \text{JS}^\ast \), e.g., functional arrays, integers, etc. In this section, we show how to encode the proof obligations for \( \text{JS}^\ast \) programs in a first-order theory amenable to automated proving via Z3.

#### 6.1 Incremental compilation of higher-order queries

We have seen two examples of higher-order formulas so far. The first case arose when verifying the \( \text{JSPrims} \) library, in particular in the implementation of \text{apply} in §4.3. There, in the first branch of the match (at line 7), we had to prove that a context \( \Gamma_1 = \Gamma_0 \cdot \text{dyn} \cdot \text{args} \cdot \text{dyn} \cdot \text{this} \cdot \text{dyn} \cdot h_0 \cdot \text{heap} \cdot (\text{f=undefined } \lor \ldots \lor \text{f=Obj}) \). \text{Unfun } (\text{TypeOf } f) \text{ args } \Rightarrow \text{Post } h \cdot \neg (\text{TypeOf } f \Rightarrow \text{Function } \text{FooTX}) \Rightarrow \text{Unfun } (\text{TypeOf } f \Rightarrow \text{args } \Rightarrow (\text{fun } \Rightarrow (\phi_2))) \cdot \text{is} \text{ unsatisfiable}. Our idea is to implement a query compiler that translates higher-order problems to a series of first-order problems that Z3 can understand. We use the program \( \Gamma_1 \) to illustrate—the same strategy applies to \( \Gamma_1 \) and \( \Gamma_2 \).

Rather than present \( \Gamma_1 \) directly to \( Z3 \), our query compiler implements a simple, incremental, first-order solving strategy outside of \( Z3 \). We describe our algorithm as an iterative process, arguing for its soundness at each step.

**Step 0: Handling a first-order theory.** If the query is first-order, then prove it unsatisfiable in \( Z3 \). If this fails, reject the program.

**Step 1: Extract a first-order partial theory.** The query compiler begins by identifying a first-order subset of the theory. In our example, this means translating \( \Gamma_1 \) into \( \Gamma_0 \cdot (\text{TypeOf } f \Rightarrow \text{Function } \text{FooTX}) \), leaving a residue \( \neg (\text{Unfun } (\text{TypeOf } f) \Rightarrow (\text{fun } \Rightarrow (\phi_2))) \), a higher-order formula to be compiled later. The theory \( \Gamma_0 \) is first-order (provided \( \Gamma_0 \) is, which it is). The goal is to find a way to evaluate the \text{Unfun } (\text{TypeOf } f) (the predicate transformer in the residue) into some concrete predicate transformer.

**Step 2: Extract a candidate witness from a model for the partial theory.** Next, we present the first-order partial theory to \( Z3 \) and attempt to prove this theory satisfiable. If \( Z3 \) decides that the partial theory is unsatisfiable, then we are done: clearly, via weakening, the whole context is also unsatisfiable. If we have a quantifier free theory, the only other possibility is for \( Z3 \) to decide that the theory
is satisfiable, and in this case Z3 produces a model $M$. If the theory has quantifiers, Z3 may reply “unknown”, but, even in this case, Z3 produces a partial model—we discuss the problem of generating models for theories with quantifiers in further detail in §6.2. Given the model $M$, we ask Z3 to evaluate the guard of the residue—in our example, this is $\neg \text{(Unfun (TypeOf f))}$—and we obtain a type $Tx$ which is a candidate predicate transformer that we can use to compile the residue of the query.

Step 3: Confirm the candidate witness. Of course, our partial theory may have many models, and our candidate $Tx$ may not be a valid solution. Our next step is to prove that $\neg \text{(Unfun (TypeOf f))}=Tx$ is true in all models, i.e., we ask Z3 to prove the unsatisfiability of $\Gamma_1, \neg \text{(Unfun (TypeOf f))}=Tx$. If this step fails, then we can proceed no further and our query compiler rejects the entire query saying that it was not able to prove the theory’s unsatisfiability. However, if this step succeeds, we have confirmed that $Tx$ is a valid witness and can move to step 4. In our example, we have $Tx=\text{FooTX}$.

Step 4: Compile the rest of the query using the witness. Using our valid witness, we can compile the residue further. In our example, we now have to prove that $\neg \text{FooTX args global (fun \_ \Rightarrow \phi_2)}$ is unsatisfiable in the theory $\Gamma_1$. But, $\text{FooTX}$ is a concrete transformer $\text{fun args this 'Post} \Rightarrow \phi_2$, and our goal now contains a redex which can be reduced via several $\beta$-reductions to a first-order formula, with potentially some higher-order residues contained within. So, we iterate the process and go back to step 0.

Termination of the query compiler. The satisfiability problem for a first-order theory is undecidable. Thus, at each call to $Z3$, our query compiler may diverge, and hence the compilation of a higher-order query into a series of first-order queries is, necessarily, incomplete.

In the future, it may be interesting to identify a fragment of higher-order logic that can be compiled to a finite sequence of effectively propositional theories, and to obtain an complete decision procedure for this case. However, the verification conditions for most common programs is unlikely to fall in this fragment.

6.2 Obtaining candidate witnesses from an external tool

Extracting candidate witnesses by generating heap models in Z3 can be expensive—we observed running times of tens of minutes for model generation in some cases. This can make the entire query compilation process extremely slow. By far the most expensive step in our query compilation procedure is the model generation in step 2. However, step 2 is only necessary to produce a candidate witness. If we can efficiently guess a candidate by some other means, then we can bypass step 2, and use step 3 directly to confirm our guess—confirming the candidate requires the use of an unannotated query, and on such queries, Z3 is typically very efficient (on the order of a few seconds, rather than several minutes). If the guess is incorrect, we fall back on model generation, but good guesses provide us with a fast path through query compilation.

Recall that we require a witness for the predicate transformer at each function call in an JavaScript program. We use Gatekeeper (Guarnieri and Livshits 2009), an efficient but generally unsound pointer analysis on JavaScript programs, to produce a may-alias set for the target of function calls. We carry this information in the JS2JS* translation and use it to speed up the query compiler.

We begin by showing the signature of $\text{apply\_hint}$, a variant of the $\text{apply}$ combinator in JSPrims.

```
val apply\_hint1: 'Tx1::dyn \Rightarrow dyn \Rightarrow (dyn \Rightarrow heap \Rightarrow E) \Rightarrow heap \Rightarrow E 
-> fdyn \Rightarrow this:dyn \rightarrow args:dyn 
-> D (fun \_ h => (TypeOf f)=Function 'Tx1) & & 'Tx args this 'P h)
```

The $\text{apply\_hint1}$ function requires the caller to provide an explicit predicate transformer argument, $'Tx$, which is to serve as the witness for the specification of $f$. Unlike $\text{apply}$, the verification conditions produced for $\text{apply\_hint1}$ can be handled within the first-order fragment of Z3, without the need for expensive model generation.

To illustrate the generation of witnesses using Gatekeeper, we recall our running example from Figure 1 and show an alternative translation that makes use of hints. Gatekeeper, a flow- and context-insensitive pointer analysis, runs on the source program and concludes that the target of the call at line 2 is the function closure defined at line 1. In general, Gatekeeper provides an approximation of the call graph of the program, resolving each function call to zero or more function closures.

Translating programs with hints

```haskell
1 function foo(x) { ... }
2 foo({f:0});
3 foo = 17;
```

The JS2JS* translation makes use of this call graph to produce the JS* program above. The translation is identical to the one shown previously, except for two differences. Each definition of a lambda-term in the translated program produces an application of the $\text{Fun}$ constructor. Where previously the first argument of $\text{Fun}$ (a predicate transformer) was omitted, when a hint is available from Gatekeeper involving this function closure, JS2JS* inserts a new unification variable, 'U0 in our example, as the first argument to Fun. When the type inference algorithm computes a predicate transformer for foo, say $\text{FooTX}$, 'U0 is unified with $\text{FooTX}$.

Next, at each call site where Gatekeeper is able to definitively resolve a function call to a particular closure (say, foo), JS* inserts a call to $\text{apply\_hint1}$, passing as a first parameter the same unification variable ('U0) that was used when translating foo. As type inference proceeds and 'U0 is unified with $\text{FooTX}$, the witness argument to $\text{apply\_hint1}$ is suitably unified too.

Of course, Gatekeeper may sometimes be unable to resolve a function call definitively and can instead provide a non-empty set of potential targets for the call. We provide versions of $\text{apply\_hint}$ to support up to 10 possible call targets (of the form shown below) and the JS* translation inserts unification variables for each of the possible call targets. If Gatekeeper provides no information about a call, we resort to using the original $\text{apply}$ function.

```
val apply\_hint2: 'Tx1::dyn \Rightarrow dyn \Rightarrow (dyn \Rightarrow heap \Rightarrow E) \Rightarrow heap \Rightarrow E 
-> 'Tx2::dyn \Rightarrow dyn \Rightarrow (dyn \Rightarrow heap \Rightarrow E) \Rightarrow heap \Rightarrow E 
-> fdyn \Rightarrow this:dyn \rightarrow args:dyn 
-> D (fun \_ h => (TypeOf f)=Function 'Tx1 || TypeOf f=Function 'Tx2) & & (TypeOf f=Function 'Tx1 \Rightarrow 'Tx1 args this 'P h) & & (TypeOf f=Function 'Tx2 \Rightarrow 'Tx2 args this 'P h)
```

7. Implementation and evaluation

This section presents an empirical evaluation of applying our tool chain to verify the runtime safety of JavaScript web-browser extensions. All the major web browsers provide support for JavaScript extensions to provide a range of features for an enhanced browsing experience. Browser extensions have been the subject of some recent study, both because of their popularity, and because of the security and reliability concerns they raise (Bandhakavi et al. 2010; Barth et al. 2010; Guha et al. 2011). Extensions are executed on every page a user visits, and runtime errors caused by extensions can compromise the integrity of the entire browser platform. Thus, a methodology to prove extensions (and other JavaScript programs) free of runtime errors is of considerable practical value.

We report here on the verification of three browser extensions for the Google Chrome web browser. These extensions are based
on extensions that we studied in prior work (Guha et al. 2011). We prove each of these extensions free of runtime errors, while assuming a sequential execution model for JavaScript, rather than the asynchronous model that these extensions employ in practice, as mentioned in §1. To simplify verification, we also model collections of objects by iterators, whereas the standard DOM API provides collections of objects as arrays encoded as dictionaries. We expect to handle the array idioms directly in the future. All timing numbers were collected on a quad core 2.8GHz Windows 7 laptop.

7.1 HoverMagnifier

Our first extension is HoverMagnifier, an accessibility extension. It magnifies the text under the cursor. The core interface of HoverMagnifier with a web page is implemented in 27 lines of JavaScript.

We verified this program for the absence of runtime errors, in less than 10 seconds. In doing so, our query compiler emitted 86 queries to Z3, including resolving 6 function calls by generating Z3 models (which dominated the verification time).

A key part of its code is shown below—it involves the manipulation of a collection of DOM elements. At line 2 it calls the DOM function getElementsByTagname to get all the <body> elements in a web page. Line 3 gets the first element in the result set. Then, it checks if the body is undefined and line 6 sets an event handler, magnify, to be called whenever the user’s mouse moves—we elide the definition of magnify.

```
function magnify(evt) { ... }
var elts = document.getElementsByTagName("body");
var body = elts[0];
if (body !== undefined)
{
  body.ondmousemove = function (evt){magnify(evt);};
  body.ondmousemove(dummyEv) //added for verif. harness
```

Setting up a verification harness for such a program involves two main elements. First, we need some “driver” code to ensure that all the relevant parts of the program are exercised. For example, we add the code at line 7 to mock the firing of a mouse-move event, so that the code in magnify becomes reachable. Without this, our verification tool would still infer a weakest pre-condition for magnify, but since no call to it appears in the program, the precondition would be trivially satisfied.

```
function dummyEv() { ... }
```

Specifying the DOM (partial) using inductive predicates

```
1 type ObjField h o f = HasField h o f &
2    & TypeOf(Select h o f)object
3    & InDom h (Select h o f)

4 type IsElt :: heap ⇒ dom ⇒ E
5 type EltTyping h elt =
6    & IsElt elt = object &
7    & HasField h elt "text" &
8    & TypeOf(Select h elt "text")=string &
9    & HasField h elt "getFirstChild" &
10   & TypeOf(Select h elt "getFirstChild")=
11 function (fun args this 'Post h' ⇒ IsElt h this &
12    & Child. child=Unid || IsElt h child ⇒ "Post child h")
13 assume IsEltTrans h y h1 h2 x. {pattern (IsElt h1 x); (SelObj h2 (Loc x))}
14 (IsElt h1 x &
15   (Select h1 x "text")= SEL(Select h2 x "text") & ... &
16   (Select h1 x "getFirstChild")=SEL(Select h2 x "getFirstChild")
17   ⇒ IsElt h2 x)
18 assume IsEltTyp h x. {pattern (IsElt h x)} IsElt h x ⇒ EltTyping h x
```

More substantially, we have to provide specifications for all the APIs used by the program. For our extensions, this API is the DOM. For each kind of DOM concept (document, element, style, etc.), we define a corresponding F* type—a predicate stating that

an object is an instance of the concept. For element, a predicate EltTyping h elt means that elt is an element in heap h.

The predicate EltTyping (line 5) states that elt is an object, it has a field getFirstChild, and that this field is a function whose specification is given by the predicate transformer at lines 10–11. Informally, getFirstChild expects its first argument (the implicit this pointer) to be a DOM element e, and, if e is not a leaf node, it returns another DOM element (otherwise returning undefined).

Capturing this specification involves the use of an inductive predicate. We do this using the predicate IsElt, and then providing two assumptions (at the bottom of the display) giving an interpretation to IsElt. The assumption IsEltTrans states that IsElt is transitive in its heap argument (if the relevant fields of the element elt have not changed), and IsEltTyp expands IsElt back into EltTyping. To control how Z3 uses these quantified assumptions, we provide patterns, which serve to guide Z3’s quantifier instantiation algorithm.

```
1 type Enumerable "P h d = (TypeOf h d object) & IsElt h d &&
2    & HasField h d "next" & (TypeOf(Select h d "next")) =
3    & (function (fun args this 'Post h' ⇒ this = d &
4    & (x(dy. (>=Unid || "P h' x") ⇒ 'Post x')))
5    & DocTyping h doc = HasField h doc "getElementsByTagName"
6    & x & (TypeOf(Select h doc "getElementsByTagName")) =
7    & (function (fun args this 'Post h1' ⇒
8    & (this = doc & SingletonString h1 args &
9    & (x. Enumerable IsElt h1 x ⇒ 'Post x h1')))

10 & val global : dyn
11 & type InitialHeap predicate, with a type for the document object
12 DocTyping h global document
13 ObjField h0 global "document" &
14
```

The display above shows the predicate DocTyping, a partial specification for the document object (line 6). It states that the object doc contains a function-typed field "getElementsByTagName". The pre-condition for this function requires that it be called with its argument set to the doc object itself. All JavaScript functions receive their enclosing objects as their this parameter. However, since functions are first-class and can be stored within other objects, statically predicting the this pointer of a function is non-trivial. For example, in the following program, the final function call receives the object e as the this parameter: o. f = document.getElementById & f = singletonString el. This can be problematic, and, in the case of the DOM, leads to a runtime error. We rule out this kind of error by requiring that every call to getElementsByTagName must pass this parameter equal to the document object doc.

The pre-condition of getElementsByTagName also requires that its arguments object args contain a single string field (the predicate SingletonString, elided here for brevity). The post-condition of getElementsByTagName is captured by line 10. It states that the function does not change the heap, and that the object x returned satisfies the predicateEnumerable IsElt h1 x.

The predicate Enumerable is shown at line 1. It is parameterized by a predicate "P" that applies to each of the elements in the collection. Enumerable collections are objects that have a function-typed "next" field which does not mutate the heap. The function either returns Undefined (if the collection is exhausted), or returns a value satisfying the predicate "P h' x". As with other functions, "next" expects its this pointer to be the enclosing collection.

Finally, we extend the InitialHeap predicate (shown first in §6.2) to include the assumption about the document object.

7.2 Facepalm

Our next example is Facepalm, an extension that helps build a user’s address book by automatically recording the contact infor-
ation of a user’s friends as they browse Facebook. It is implemented in 87 lines of JavaScript. Overall, verifying the entire extension took about 6 minutes. The verification time was dominated by the time spent in Z3. Our query compiler asked about 1,300 Z3 queries, and required producing models to resolve 27 function calls. Gatekeeper was able to successfully provide us with a hint 11 times, but the remaining 16 times we fell back on Z3’s model finding feature, which dominated the Z3 time—so, a reduction in the number of queries that require producing models (via better hints) is likely to reduce the verification time substantially. We expect better stubs of queries that require producing models (via better hints) is likely to reduce the verification time substantially. We expect better stubs for the DOM when configuring Gatekeeper to help.

The main function of Facepalm is shown below (start at line 16). At a high level, this extension checks to see if the page currently being viewed is a Facebook page (line 18). If the check succeeds, it traverses the DOM structure of the page looking for a specific fragment that mentions the name of the user’s friend (line 19). A second traversal finds the friend’s contact and website information (line 20). If this information is successfully found, the extension logs it and saves it to the user’s address book maintained on a third-party bookmarking service (line 24).

The main function and a DOM traversal function in Facepalm

```javascript
function getPath(root, p) {
  var cur = root; var path = p;
  // needs a loop invariant
  while (path !== undefined && cur !== undefined) {
    cur = cur.getChild(path.hd); // needs a hint
    path = path.tl;
  }
  return cur;
}
```

At line 15, we show the signature of `getPath` and see how the function gets called at line 16. The fifth argument is the thunk representing the loop guard, and the fourth argument is the thunk representing the loop body. This means, e.g., manually.

The main interest in verifying Facepalm is in verifying the two DOM traversals, findName and findWebsite. Both of these involve `while`-loops to iterate over the structure of the DOM. They do this by eventually calling the function `getPath`, shown at line 1. The loop in `getPath` iterates simultaneously over a list (path) of integers as well as the DOM tree rooted at `cur`, where the integer in the list indicates which sub-tree of `cur` to visit. Function `getChild(n)` returns the `n`th child of an element.

To verify this code, the programmer needs to supply a loop invariant. At the moment, JS2JS has only primitive support for annotating JavaScript source with loop invariants and other specifications. Thus, we describe the verification at the JS’ level.

We start by showing (in the next display) two combinators in our `JSPrims` library used in the translation of `while`-loops. The signature of `while` shows a function that takes three predicate parameters, a loop guard, and a loop body. Its implementation iterates the application of `body` so long as `guard` is true, and then returns `Undefined`. The interesting element is, of course, in its specification.

The first predicate parameter to `while` is `Inv`, an invariant on the `heap`. The next two parameters are predicate transformers specifying the loop `guard` and `body`, respectively. While our verification condition generator can infer an instantiation for `TxGuard` and `TxBody`, the loop invariant `Inv` has to be supplied by some other means, e.g., manually.

Of course, `Inv` has to be an inductive invariant—the predicate transformer for `while` states this as a pre-condition. At line 7, we state that the invariant must hold on the initial heap. At line 8, we say that the invariant must be inductive—this may take some careful study to understand, since it makes heavy use of the composition of predicate transformers. Informally, it states that for any heap `h1` that satisfies the invariant, then if the loop guard executed in the heap `h1` returns `true`, then running the loop body re-establishes the invariant `Inv`; otherwise, if the loop guard returns `false`, the invariant `Inv` must be true again. The final condition (line 13) requires that `Inv` be sufficient to establish any post-condition of the loop.

The `JsPrims` combinators for `while`-loops and for getting the heap

```javascript
val _while: 'Inv :: heap ⇒ E
  → TxGuard:(bool ⇒ heap ⇒ E) ⇒ heap ⇒ E
  → TxBody:(dyn ⇒ heap ⇒ E) ⇒ heap ⇒ E
  → guard(unit ⇒ DST bool "TxGuard"
    → body(unit ⇒ DST dyn "TxBody")
    → DST dyn (fun 'Post h ⇒
      'Inv h &
      (if 'h1. 'Inv h1
       → ('h1 & 'Inv h1)
       → (Inv implies Post))
    )
  )
  h0

val get: unit ⇒ DST heap (fun 'Post h ⇒ 'Post h h)
```

At line 15, we show the signature of `get`, a simple function that returns the current heap as a value. As we will see, this is useful for stating invariants.

Now, we return to analyzing `getPath` and seeing loop invariants for clients of `while` in action. The next listing shows the translation (slightly cleaned up) of `getPath` to E`*` for verification. At lines 2–4, we initialize the two local variables corresponding to `cur` and `path` in the source program. The while loop is translated to a call to the `while` combinator. The first three arguments to `while` (at line 7) are the predicate arguments—the first argument `Inv` locals `h0` is provided by the programmer; the next two are wild cards (\(\_\)) whose instantiation is inferred by the verification condition generator. The fourth argument is the thunk representing the loop guard, and the last argument is a thunk for the loop body.

Translation of `getPath` to JS

```javascript
let getThis = arguments[0] in
let locals = allocObject () in
update locals "cur" (select locals "0" in)
update locals "path" (select locals "1" in);

let h0 = getThis in
let _ = while (Inv locals h0) _
  (fun _ ⇒ (not (select locals "path" = Undefined)) &&
    (not (select locals "cur" = Undefined)))
  (fun _ ⇒
    let params = allocObject () in
    let getChild = select locals "cur" "getChild" in
    let hd = select locals "path" "hd" in
    update params "0" hd;
    update locals "cur" (apply getChild (select locals "cur" params))
    update locals "path" (select locals "path" "tl") in
    select locals "cur"
```

The code of the loop guard is straightforward. The body allocates an object `params` to pass arguments to the function `getChild`. 
The parameters at the call on line 15 is a singleton integer containing the head of the list "path". We then update the locals "cur" and "path" and iterate.

Intuitively, verifying this code for the absence of runtime errors requires two properties: at each iteration, the "path" local must either be Undef or contain an integer hd field and also a tl field, while the "cur" local must contain a DOM element (or be Undef). We state just this using the loop invariant shown in the next display.

The invariant comes in two parts. First, at lines 2–14 we define an inductive predicate IsList a h l, which states that in the heap h, the value i is either Undef or a list of 'a-typed values. The style of this inductive specification is similar to the specification we used for IsElt in §6.2. The invariant Inv itself is defined at line 18. The invariant is a ternary predicate relating an object holding the local variables locs, the heap h0 at the start of the loop, to a heap h1, which represents the heap at the beginning of each loop iteration.

The invariant for getPath

The invariant states that: (1) the loop does not mutate the global object (necessary to verify the rest of the program after the loop); (2) the locs object has fields "path" and "cur"; (3) the former is a list of integers; and (4) the latter is either Undef or a DOM element. Stating and proving (4) required an additional hint for Z3. Instead of simply writing IsElt, we had to write CutsElt, which provides Z3 with a lemma to prove first that the "cur" local contains an object, and using that lemma, prove that it is a DOM element. The lemma helps guide the pattern-based instantiation of the quantifiers needed to reason about the IsElt predicate.

Clearly, writing such an invariant took considerable manual effort. This is unsurprising—verifying loops in a more well-behaved language, say, C#, also requires writing invariants, although, of course, many simple invariants in C# can be stated using just its type system. With more experience, we hope to discover JavaScript idioms that make writing loop invariants easier, and further, to apply ideas ranging from abstract interpretation to interprocedural to automatically infer these invariants.

7.3 Typograf

Our final example is Typograf, an extension that formats text a user enters in a web form. Typograf, like most Chrome extensions, is split into two parts: content scripts which have direct access to a web page, and extension cores which access other resources. The two parts communicate through message passing. When Typograf’s content script receives a request from the extension core to capture text, it calls captureText, which calls the callback function in the request (line 2). At line 8, Typograf registers listener as an event handler with the Chrome extension framework, by calling the function addListener. We show a simplified fragment of its code below.

Message passing with callbacks in Typograf

8. Related work

Our JS2JS* tool is based closely on the λJS tool Guha et al. (2010). We began our implementation based on the λJS code base. However, we proceeded to reimplement a λJS-like translation from scratch, for two reasons. First, we found the original λJS had a brittle parser—parsing JavaScript is not easy (mainly because of its
unusual semi-colon and off-side rule), and λJS failed to successfully parse many programs we were interested in. Second, in order to exploit translation hints from Gatekeeper, a reimplementation within the framework of Gatekeeper was more convenient.

Our core verification methodology is connected to a long line of literature of Hoare logic, as well as dependently typed programming languages—we have discussed this connection already. Here, we focus mainly on theories and analyses for dynamically typed programming languages in general, including JavaScript.

There is a long tradition of aiming to equip dynamic languages with a notion of static typing, starting perhaps with Cartwright (1976). More recently, Henglein and Rehof (1995) defined a translation from Scheme to ML by encoding Scheme terms using the algebraic ML type dynamic. Our work is related to this line. First, our JS2JS* translation makes use of a similar Scheme to ML translation (combined with λJS). Second, Henglein and Rehof were also able to statically discover certain kinds of runtime errors in Scheme programs via their translation to ML. Our methodology also aims to statically detect errors in dynamically typed programs via a translation into a statically typed language. Because of the richness of our target language, we are able to verify programs in a much more precise (and only semi-automated) manner. Besides, we need not stop at simply proving runtime safety—our methodology enables proofs of functional correctness.

There are several other recent systems for dynamic typing based on dependent typing. Dminor (Bierman et al. 2010) provides semantic subtyping for a first-order dynamically typed language. Tobin-Hochstadt and Felleisen (2010) provide refinement types for a pure subset of Scheme. System D (Chugh et al. 2012) is a refinement type system for a pure higher-order language with dictionary-based objects. Our type dyn bears some resemblance to these in that the refinement formulas speak about a typing property. However, none of these prior systems gives stateful refinements to functions, which prevents them from handling both higher-order functions and mutable state, as we do.

Gardner et al. (2012) provide an axiomatic semantics for JavaScript based on separation logic. Their semantics enables precise reasoning about first-order, eval-free JavaScript programs, including those that explicitly manipulate scope objects and prototype chains. Technically, supporting this idiom is possible in our system with a richer heap model (as in our concurrent submission). However, automated proving for such complex idioms is still hard. Indeed, at present, Gardner et al. provide only pencil and paper proofs about small, first-order JavaScript programs. Nevertheless, a potential direction for future work is to embed a subset of Gardner et al’s separation logic style within $F^\star$ for JavaScript verification.

In other work Gardner et al. (2008) show how to write specifications and reason about the DOM using context logic. Our specification of the DOM, in contrast, uses classical logic, and is not nearly as amenable to modular reasoning about the DOM, which has many complex aliasing patterns layered on top of a basic n-ary tree data structure. Understanding how to better structure our specifications of the DOM, perhaps based on the insights of Gardner et al., is another line of future work.

Many tools for automated analyses of various JavaScript subsets have also been constructed. We have already mentioned Gatekeeper, a pointer analysis for JavaScript used by JS2JS*. The CFA2 analysis (Vardoulakis and Shivers 2011) has been implemented in the Doctor JS tool to recover information about the call structure of a JavaScript program. Our method of reasoning about JavaScript programs by extracting heap models in $\mathcal{Z}$ can also be seen as a precise control flow analysis. As discussed previously, there is ample opportunity to improve our tool to consume the results of a source-level control-flow analysis as hints to our solver.

Jensen et al. (2009) build a whole-program abstract interpretation to recover more precise type information of programs in order to statically find program errors. The goals are similar to our work, but our modular approach does not rely on complete program analysis. They assume relatively small, complete programs while our approach allows for modular verification of portions of a program. Additionally, our analysis provides a more generalized approach to verifying policies on a program rather than just searching for program errors; in this work, we just happen to verify a policy of no basic program errors.

Conclusions. JavaScript has a dubious reputation in the programming language community. It is very popular, but its semantics is considered so unwieldy that sound, automated analysis is believed extremely hard. Our work establishes that with the right abstractions for reasoning about higher-order, dynamically typed stores, automated program verification tools are within reach. With our current prototype implementation, we are able to verify several JavaScript browser extensions for the absence of runtime errors. Our results open the door to applying a wealth of research in automated program verification techniques to JavaScript.

References


