Monadic Refinement Types for Verifying JavaScript Programs

Nikhil Swamy[1] Joel Weinberger[2] Juan Chen[1] Ben Livshits[1] Cole Schlesinger[3]

Microsoft Research[1] UC Berkeley[2] Princeton University[3]

Abstract

Researchers have developed several special-purpose type systems and program logics to analyze JavaScript and other dynamically typed programming languages. Still, no prior system can precisely reason about both higher-order programs and mutable state; each system comes with its own delicate soundness proof (when such proofs are provided at all); and tools based on these theories (when they exist) are a significant implementation burden.

This paper shows that JavaScript programs can be verified using a general-purpose verification tool—in our case, F⋆ (Swamy et al. 2011), a dependently typed dialect of ML. Our methodology consists of a few steps. First, we extend prior work on λJS (Guha et al. 2010) by translating JavaScript programs to F⋆. Within F⋆, we type *pure* JavaScript terms using a *refinement* of the type dyn, an algebraic datatype for dynamically typed values, where the refinement recovers more precise type information. *Stateful* expressions are typed using the Hoare state monad. Relying on a general-purpose weakest pre-condition calculus for this monad, we obtain higher-order verification conditions for JavaScript programs that can be discharged (via a novel encoding) by an off-the-shelf automated theorem prover. Our approach enjoys a fully mechanized proof of soundness, by virtue of the soundness of F⋆.

We report on experiments that apply our tool chain to verify a collection of web browser extensions for the absence of JavaScript runtime errors. We conclude that, despite commonly held misgivings about JavaScript, automated verification for a sizable subset of the language is feasible. Our work opens the door to applying a wealth of research in automated program verification techniques to JavaScript programs.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Syntax and Semantics

General Terms Verification, Languages, Theory

Keywords Type systems, refinement types, Hoare logic, dynamic languages

1. Introduction

JavaScript is the lingua franca of the web. Its highly dynamic nature contributes both to the perception that it is easy to use (at least for small programs), as well as to the difficulty of using it to build secure applications. Whereas traditionally the security threat has been limited by executing JavaScript within the confines of a web browser's sandbox, the scope of JavaScript is now considerably broader, and recent trends indicate that it is likely to broaden further still. For example, Metro[1] applications in the forthcoming Windows 8 operating system can be programmed in JavaScript, and the language can be used to perform a wide range of system calls on the host operating system. Tools that can effectively analyze JavaScript for attacks and vulnerabilities are thus becoming a pressing need.

[1] `http://msdn.microsoft.com/en-us/windows/apps/`

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

```
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 MLjs version

Reasoning about JavaScript programs, however, poses several challenges. Specifying the semantics of JavaScript is itself a significant research question with which several groups are currently engaged (Guha et al. 2010; Herman and Flanagan 2007; Maffeis et al. 2008). Rather than address this again, we adopt Guha et al.'s solution. Their approach, λJS, gives a translation of a sizeable fragment of JavaScript into a Scheme-like dynamically typed lambda calculus. Based on λJS, we developed a tool JS2ML, which translates JavaScript programs to ML. JS2ML, in effect, composes the λJS translation with a standard Scheme to ML translation. What remains then is merely to reason about the programs emitted by JS2ML.[2] However, as one might expect, several difficulties remain.

1.1 A challenge: dynamically typed higher-order stores

The semantics of JavaScript involves many features and subtle interactions among the features. These include prototype hierarchies, scope objects, implicit parameters, implicit conversions, etc. However, following λJS, the JS2ML translation simply desugars these features into standard ML constructs.

Figure 1 illustrates the JS2ML translation—we call programs in the image of the JS2ML translation "MLjs programs". The functions allocObject, select, and update are functions defined in a library for MLjs programs—they serve to create heap-allocated objects and to read and modify their fields. This library also defines a type dyn for dynamically typed values, with constructors like Str, Int, and Fun to inject values into the dyn type.

The definition of the JavaScript function `foo` is translated to the λ-term foo in ML, 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 are dictionaries indexed by string-typed keys, rather than

[2] While Guha et al. demonstrate that their semantics is faithful using an extensive test suite, they make no claims that the λJS translation is fully abstract. Thus, when analyzing resulting programs of the λJS translation, we might miss some behaviors of the source program. This is a limitation of our current work which we seek to address in the future by using a higher fidelity translation.

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, 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 v, for some v), and passing two arguments—the global object, the receiver object for the call; and the args object containing a single argument. Finally, 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, *every* function call is done in this style.

1.2 Contributions

We provide a new method for verifying dynamically typed, higher-order stateful programs via a translation into F⋆ (Swamy et al. 2011), a dependently typed dialect of ML. Our contributions are as follows.

−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 (Henglein 1994), which combines dynamic typing with the Hoare state monad (Nanevski et al. 2008). We verify the implementation of this library against a strong specification, ensuring assertion safety for well-typed clients of this library. (§3)

−We develop JS2ML, a translation from JavaScript to ML, and type the resulting (MLjs) programs against the JSPrims API for verification. (§6.1)

−We provide a (semi-)automated methodology for verifying clients of JSPrims. Our approach involves the use of a general-purpose verification condition generator (VCGen) for ML programs (in F⋆), which, given loop invariants, infers a weakest pre-condition for a program. The VCGen is described in detail and proved sound in a concurrent submission (Schlesinger and Swamy 2011). In this paper, we demonstrate that our VCGen can be effectively applied to MLjs programs. (§4)

−The proof obligations produced by VCGen make heavy use of higher-order logic. Our fourth contribution is a new technique that shows how higher-order verification conditions (particularly those of the form produced by VCGen) can be incrementally compiled 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) in our experiments. (§5)

−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 properties of recursive data structures (like DOM elements) using inductive predicates, and giving specifications to higher-order functions (including in some cases, functions of the third order). (§6)

−We evaluate our work experimentally by verifying a collection of JavaScript web-browser extensions (on the order of 100 lines each) for various properties, including, principally, the absence of JavaScript runtime errors. In each case, apart from carefully writing loop invariants, verification was automatic. (§6)

1.3 Limitations

Our work also suffers from two broad classes of limitations, first, related to the fragment of JavaScript we currently support, and, second, the inefficiency of our current prototype.

JavaScript fragment. Similar to other work, we consider closed, eval-free JavaScript programs. We assume a sequential control flow—we view the treatment of asynchrony in JavaScript as a challenging, but orthogonal, problem. We provide limited support for the mutation of prototype chains after an object is created—handling more general prototypes is feasible with an additional level of indirection in our model of the JavaScript heap, but the proof burden is likely to be substantially larger, and the payoff for this complexity is questionable. We also provide only limited support for JavaScript arrays, instead requiring collections of objects to be handled using iterators—applying the vast body of research into verifying array-manipulating programs to JavaScript is future work.

Scaling our verification technique. At the moment, our prototype is slow. Our largest program, about 90 lines of JavaScript source, with about 10 functions, and two loops took about 70 minutes to verify. The majority of this time is spent within Z3, particularly in its partial model finding feature, which has not at all been tuned for the kinds of problems we generate—we expect to tune Z3 for our purposes in the immediate future. Furthermore, we show that with some simple hints provided by the translation, performance can be improved dramatically.

2. A review of F⋆ and the Hoare state monad

We begin by presenting a primer on F⋆, in particular our encoding of the Hoare state monad. For a more thorough presentation consult Swamy et al. (2011).

F⋆ is a variant of ML with a similar syntax and dynamic semantics but with a more expressive type system. It enables general-purpose programming with recursion and effects; it has libraries for concurrency, networking, cryptography, and interoperability with other .NET languages. After typechecking, F⋆ is compiled to .NET bytecode, with runtime support for proof-carrying code. It can also be compiled to JavaScript, via type erasure. It has been used to program and verify more than 40,000 lines of code, including security protocols, web browser extensions, and distributed applications. Its main typechecker, compiler, and runtime support are coded in F#. We have also developed a core typechecker programmed in F⋆ itself that has been self-certified for correctness (Strub et al. 2012).

The main technical novelty of F⋆ is a kind system to keep track of different fragments of the language and control their interaction. By default, program terms (with kind $\star$) may have arbitrary effects. Within these programs, proof terms (with kind P) remain pure and terminating, and thereby provide a logically consistent core. Besides, affine resources (with kind A) may be used for modular reasoning on stateful properties. Finally, ghost terms (with kind E) may be used only in specifications, to selectively control the erasure of proof terms, when the proofs are irrelevant or unavailable (for instance, when calling legacy code or using cryptography). A sub-kinding relation places the kinds in a partial order $P \leq \star \leq E$, with A being unrelated to the others.

In this paper, aside from the types of ML, we limit ourselves to a few main F⋆-specific typing constructs. First, we use ghost refinement types of the form x:t{ϕ}, the type of values v:t for which the formula $\phi[v/x]$ is derivable from the hypothesis in the current typing context. Second, we use dependent function arrows x:t →t', the type of functions whose domain contains values x of type t and whose co-domain is of type t', where t' may depend on x. We also use function arrows whose domain are types 'a of kind κ and

whose codomain are types t, written 'a::k → t, i.e., these are types of functions polymorphic in types 'a of kind k.

The only base kinds we use in this paper are $\star$ and E. We also use the product kinds k ⇒ k' and t ⇒ k for type constructors or functions. For example, the list type constructor has kind $\star \Rightarrow \star$, while the predicate Neq (for integer inequality) has kind int ⇒ int ⇒ E. We also write functions from types to types using the notation **fun** ('a::k) ⇒ t, and from terms to types using **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 $\star$ 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 specificational), 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, 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 type 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 Hoare state monad. F* supports several idioms of programming with effects. One idiom, mentioned briefly earlier, involves the use of affine types for resource usage. Another mechanism adopted in F* is the Hoare state monad, shown below.

Encoding the Hoare state monad in F*

```
type heap :: ⋆
type ST ('Pre::heap ⇒ E) ('a::⋆) ('Post::'a ⇒ heap ⇒ E) =
     h:heap{'Pre h} →(x:'a ∗ h':heap{'Post x h'})

val unitST : 'a::⋆
             →'Post::'a ⇒ heap ⇒ E
             →x:'a
             →ST ('Post x) 'a 'Post
let unitST 'a 'Post x h = (x,h)

val bindST : 'a::⋆ →'b::⋆
           →'Pre::heap ⇒ E
           →'Post1::'a ⇒ heap ⇒ E
           →'Post::'b ⇒ heap ⇒ E
           →ST 'Pre 'a 'Post1
           →(x:'a →ST ('Post1 x) 'b 'Post)
           →ST 'Pre 'b 'Post
let bindST 'a 'b 'Pre 'Post1 'Post f g h = let x, h' = f h in g x h'
```

Informally, the Hoare state monad is simply the state monad (heap →('a ∗ heap)) augmented with predicates to track the pre- and post-conditions of a stateful computation that may read or write a heap and produce an 'a-typed result. In the listing above heap is an abstract type, and the ST 'Pre 'a 'Post is the type of computation which when run in a heap h satisfying the pre-condition 'Pre h produces a result x:'a and an output heap h' satisfying the relation 'Post x h, unless it diverges.

This parameterized monad comes with the usual unit and bind operations. The signature of the unitST function shows that any value x:'a can be injected into the monad to produce a computation that can satisfy any post-condition 'Post::'a ⇒ heap ⇒ E, so long as the input heap satisfies 'Post x. The bindST function allows two stateful computations to be composed, so long as the post-condition of the first matches the pre-condition of the second.

Notice that writing specifications for the ST-monad that are polymorphic in the post-condition predicate 'Post is quite convenient, e.g., one does not have to directly state that unitST leaves the input heap h unchanged. Without this polymorphic style one would need state extra conditions to relate the output heap to the input heap, for example, by making use of two-state predicates as post-conditions. To avoid the additional complexity of two-state predicates (and to simplify type inference), we embrace post-condition polymorphism wholeheartedly—throughout this paper, we use a variant of the Hoare state monad proposed by Schlesinger and Swamy (2011) and shown below.

The post-condition polymorphic Hoare state monad

```
type DST ('a::⋆) ('Tx::('a ⇒ heap ⇒ E) ⇒ heap ⇒ E) =
     ('Post::'a ⇒ heap ⇒ E) →ST ('Tx 'Post) 'a 'Post
val returnDST: x:'a →DST 'a (fun ('Post::'a⇒ heap⇒ E) ⇒ 'Post x)
val bindDST: 'Tx1::('a ⇒ heap ⇒ E) ⇒ heap ⇒ E
           →'Tx2::'a ⇒ ('b ⇒ heap ⇒ E) ⇒ heap ⇒ E
           →DST 'a 'Tx1
           →(x:'a →DST 'b ('Tx2 x))
           →DST 'b (fun ('Post::'b⇒ heap⇒ E) ⇒
                    'Tx1 (fun (x:'a) ⇒ 'Tx2 x 'Post))
```

The DST 'a 'Tx type is a polymorphic Hoare state monad, where 'a is the type of the result of the computation and 'Tx is a *predicate transformer* (Dijkstra 1975), which, given any post-condition, computes a (usually weakest) pre-condition on the input heap which ensures that the computation can be executed successfully.

Since DST is defined in terms of ST, the binds and units of the ST monad carry over naturally. However, a more direct formulation can also be given—their signatures are shown above. Observe that bindDST differs from bindST in that there is no need for the post-condition of the first computation to precisely match the pre-condition of the second—since post-conditions are always polymorphic, a pre-condition for the entire composed computation can be computed simply by composing their predicate transformers. This should convey some intuition for why type inference is easier with DST rather than ST—no subsumption rule to strengthen pre-conditions and weaken post-conditions is necessary and a verification condition on the input heap can simply be computed using unification and function composition.

3. A library for dynamic typing in F*

We now describe an F* library, JSPrims, which defines various constructs suitable for verifying MLjs programs.

3.1 A refined type dynamic

Figure 2 shows the definition of our type dyn, based on the standard algebraic type dynamic, but with refinement types on each case to recover precision. For example, we have the constructor Int, which allows an integer i to be injected into the dyn type. The type of (Int i) is d:dyn{EqTyp (TypeOf d) int}. (The type of Str is similar.) The refinement formula uses the binary predicate EqTyp, interpreted in the refinement logic as an equivalence relation on types. It also uses the type function TypeOf, treated as an uninterpreted function from dyn-typed values to E-types in the refinement logic. (Recall, the sub-kinding relation allows $\star$-kinded types to be promoted to E-kind.) Since EqTyp and TypeOf have special meaning in the refinement logic, we tag them with the **logic** keyword just as documentation—from the perspective of the type system, these are simply type functions.

We use the Obj constructor to promote heap-resident MLjs objects to the dyn type—the type loc, isomorphic to the natural numbers, is the type of heap locations. The Undef constructor is for the `undefined` value in MLjs.

```
type undef :: ⋆
logic type EqTyp :: E ⇒ E ⇒ E
logic type TypeOf ('a::⋆) :: 'a ⇒ E
type dyn =
  | Int : int →d:dyn{EqTyp (TypeOf d) int}
  | Str : string →d:dyn{EqTyp (TypeOf d) string}
  | Obj : loc →d:dyn{EqTyp (TypeOf d) object}
  | Undef : d:dyn{EqTyp (TypeOf d) undef}
  | Fun : 'TxF::dyn ⇒ dyn ⇒ (dyn ⇒ heap ⇒ E) ⇒ heap ⇒ E
          →(this:dyn →args:dyn →DST dyn ('TxF args this))
          →d:dyn{EqTyp (TypeOf d) (Function 'TxF)}
and Function :: (dyn ⇒ dyn ⇒ (dyn ⇒ heap ⇒ E) ⇒ heap ⇒ E) ⇒ E
```

Figure 2. A refinement of type dynamic

The case for Fun merits closer attention. As we will see, the JS2ML translation translates every JavaScript function to a 2-ary, 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. So, to first approximation, the type of Fun is (dyn →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, for example its effects on the heap—the predicate transformers of the DST monad come to mind. We interpret every MLjs function in the DST monad, and give them types of the form this:dyn →args:dyn →DST dyn ('Tx args this), for some predicate transformer 'Tx. Then, the refinement on the Fun constructor simply records the predicate transformer capturing the semantics of its argument. The type constructor Function serves simply to coerce the predicate transformer to kind E. In contrast to the uninterpreted function TypeOf and the equivalence relation EqTyp, the types Function, int, object, etc. are treated as distinct type constants in the refinement logic. To document this difference, we tag TypeOf and EqTyp with the **logic** keyword.

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 concision and clarity. We also use the equality operator and write t = t' instead of EqTyp t t'.

3.2 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⋆ as a two-dimensional finite map from locations and field names to heap cells, this method is quite poorly suited to automated reasoning. Instead, we aim to rely on Z3's efficient theories of functional arrays to model the heap. For this, we exploit an F⋆ feature that allows embedding functions (whether interpreted or not) into the refinement logic. In our case, we embed functions corresponding to the select and update functions of Z3's array theory. As a simplifying assumption, at this stage of our work we require that JavaScript programs never mutate an object's prototype chain after the object has been created. We discuss this further in §6.1.

Figure 3 shows our internal model of the heap. MLjs programs that are clients of this library *cannot* directly manipulate any of the programmatic elements of this model (e.g., directly call the function selcell 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

```
1  type heapcell = Cell : 'a::E →d:dyn{(TypeOf d)='a} →heapcell
2  (* Selecting and updating all the fields of an object *)
3  logic type fields
4  logic val SelObj : heap →dyn →option fields
5  logic val UpdObj : heap →dyn →fields →heap
6  logic val Emp : fields
7  (* Selecting a single field *)
8  type fn = string (* field names *)
9  logic val SelCell : heap →dyn →fn →option heapcell
10 private val selcell : h:heap →o:dyn{(TypeOf o)=object} →f:fn
11         →o:option heapcell{o=SelCell h o f}
12 (* Updating a single field *)
13 logic val UpdCell : 'a::E →heap →dyn →fn →dyn →heap
14 private val updcell : 'a::E →h:heap →o:dyn{(TypeOf o)=object}
15 →f:fn →d:dyn{(TypeOf d)='a} →h':heap{h'=(UpdCell 'a h o f d)}
16
17 (* Derived constructs *)
18 (* Selecting the value in a heap cell *)
19 logic val Select : heap →loc →fn →dyn
20 assume ∀h l f v. (SelCell h l f=Some (Cell _ v)) ⇒ (Select h l f = v)
21 assume ∀h l f. (SelCell h l f=None) ⇒ (Select h l f = Undef)
22 (* Selecting the type in a heap cell *)
23 logic type SelectT :: heap ⇒ loc ⇒ fn ⇒ E
24 assume ∀h l f 'a. (SelCell h l f=Some (Cell 'a _)) ⇒ (SelectT h l f = 'a)
25 assume ∀h l f. (SelCell h l f=None) ⇒ (SelectT h l f = undef)
26 (* Checking if a heap cell is allocated *)
27 type HasField = fun h l f ⇒ (SelCell h l f)=(Some _)
28 (* Checking if an object is present in the heap *)
29 type InDom = fun h d ⇒ ((SelObj h d) <> None)
30 (* Allocating a new object *)
31 logic val New : heap →dyn →option fields →heap
32 assume ∀h d f. not (InDom h d) ⇒ (New h d (Some f))=(UpdObj h d f)
33 val alloc: proto:dyn{(TypeOf proto)=object && InDom h proto}
34         →h:heap →(x:dyn * h':heap{h'=New h d (SelectObj h proto)})
```

Figure 3. An internal model of the JavaScript heap

(TypeOf d)='a. The whole heap is essentially a two-dimensional array of heapcell, where the array is indexed first by objects and then field names. We start by defining an abstract type fields, which is interpreted in the logic as a partial map from fieldnames (fn) to heapcells. The function SelObj h d returns all the fields (if a set of fields is present) of the object d, while UpdObj h d f updates object d's fields with f—both these functions are interpreted via the standard select/update axioms of functional arrays. The value Emp is the empty set of fields. The **logic** keyword preceding each of these definitions indicates that these are only usable in F⋆'s refinement logic, not in the executable part of the program—MLjs programs never access all the fields of an object *en masse*.

At lines 7–15 we show interpreted functions in Z3, SelCell and UpdCell, to select and update individual heap cells. As with SelObj and UpdObj, these functions are not available in executable code. However, real JavaScript programs must, of course, be able to actually read from and write to heap cells. For this, we provide two function symbols, selcell and updcell, whose implementation can be in native code, and assert that the specification of these primitive operations corresponds to SelCell and UpdCell operations. Since these functions will be called from MLjs programs and our library, we give them refined types to restrict their usage. For example, both selcell and updcell insist that they be called only with objects, not arbitrary dyn-typed values.

Based on these primitives, we define several derived constructs useful for writing specifications. The function Select selects just the value part of a heap cell, while SelectT selects just the type part. HasField is a derived predicate that checks if a heap cell is allocated, while InDom checks if an object is present in the heap.

Finally, New is our specificational function for allocating a new object in the heap. We assume an external implementation of an object allocator (alloc) and require it to satisfy the signature shown at line 33. Notice that alloc takes an object proto as an argument—all the fields of the proto argument are copied directly to the newly allocated object. This is different than in JavaScript, where prototype fields are accessed via the indirection of a reference. As pointed out in §1.3, this is a limitation of our current approach, which we adopt to simplify our heap model. For our approach to be sound, we require an object used as a prototype to be immutable—it is easy to check this immutability condition within our framework. To avoid cluttering the remainder of our presentation with this side condition, we simply assume the existence of a function allocObject which constructs a new object initialized with the fields of the default prototype `object.prototype`.

3.3 The public API of JSPrims

We now discuss 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 MLjs 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 MLjs 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, say, an int-typed value 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))=(SelectT 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 _ →(match selcell h o 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 scrutinee 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 scrutinee 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 scrutinee 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 against its specification.

Safe field update. Informally, for o:dyn, f:fn, v:dyn, the term update o f v corresponds the 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 o) &&
3  'Post Undef (Update (TypeOf v) h o f v))
4 val update: o:dyn →f:fn →v:dyn →DST dyn (UpdTx o f v)
5
6 let update o f v 'Post h = match v with
7 | Undef →(Undef, updcell undef h o f v)
8 | Str _ →(Undef, updcell string h o f v)
9 | Int _ →(Undef, updcell int h o f v)
10 | Obj _ →(Undef, updcell obj h o f v)
11 | Fun 'Pre _ →(Undef, updcell (Function 'Pre) h o f v)
```

Safe function application. Informally, for f:dyn, this:dyn, 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 applications in MLjs (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 ∀'Tx. Unfun (Function 'Tx) = 'Tx
3 assume ∀'a. 'a <> (Function _) ⇒ (Unfun 'a)=(fun _ _ _ _⇒ False)
4 val apply: f:dyn →this:dyn →args:dyn
5          →DST dyn (Unfun (TypeOf f) args this)
6
7 let apply f this args 'Post h = match f with
8 | Undef | Str _ | Int _ | Obj _ →assert False
9 | Fun 'Tx fn →fn this args 'Post h
```

Intuitively, if we can ensure that (TypeOf f)=Function 'Tx for some transformer 'Tx, 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 'Tx args this 'Post h.

Stating these requirements in a form amenable to easy proving takes a little work. First, (lines 1–3) we define a type function Unfun from arbitrary E-types to types that represent predicate

transformers. The interpretation of this function (the two **assume**s) shows that when applied to a type of the form Function 'Tx, Unfun simply projects out the transformer 'Tx. Otherwise, Unfun returns the False predicate. Using Unfun, the specification of apply is easy: the predicate transformer of apply f this args is just Unfun (TypeOf 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 f this args is unsatisfiable.

The implementation of apply is straightforward. When f is not a function, Unfun (TypeOf 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, a first-order SMT solver, is infeasible—it does not understand predicate transformers. However, a novel encoding of F* verification problems in Z3 allows these higher-order problems to be compiled into a set of first-order problems which Z3 can handle. We describe this encoding in detail in §5.

4. Generating verification conditions

We employ a general-purpose VCGen designed for ML programs to produce proof obligation for MLjs programs. A full description of VCGen is out of scope for this paper. Schlesinger and Swamy (2011) formalize VCGen as a backwards style weakest pre-condition calculus for higher-order stateful programs. We briefly recapitulate their development here, and illustrate the use of VCGen on MLjs programs using a few examples.

VCGen is formalized as a two-pass type inference algorithm for ML programs. The first pass is a standard Hindley-Milner (HM) typing phase, and the second pass (the one of main interest), relies on the HM typing to compute a more precise type for a program. In practice (as in our implementation) both phases can be interleaved.

The main judgment of VCGen is written $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$. It states that in a typing context Γ, given a post-condition ψ, a predicate of kind $t \Rightarrow$ heap $\Rightarrow E$, an ML expression e can be given the type t under the pre-condition ϕ, a predicate of kind heap $\Rightarrow E$. The judgment also includes an elaboration phase: the term e is an F* expression, the elaboration of e into F*, where the elaboration simply makes explicit all type arguments, and elaborates all non-generalized **let**-bindings into monadic bindings using the bindST combinator of §2. A related judgment for values takes the form $\Gamma \vdash v : t \rightsquigarrow \mathsf{v}$, stating that an ML value v is well-typed in the context Γ at type t and is elaborated to an F* value v.

The main soundness theorem associated with VCGen states that if $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$, then in a suitably translated context, the F* expression e can be given the type ST phi t psi, the type of the Hoare state monad, where phi, t, and psi are translations of the corresponding types ϕ, t, and ψ. A similar result for values states that if $\Gamma \vdash v : t \rightsquigarrow \mathsf{v}$, then v is well-typed in F* at a type t corresponding to t. From the soundness of F*, we conclude that the pre-condition computed by VCGen is sufficient for the absence of failing assertions.

4.1 VCGen in action

We use the MLjs program in Figure 1 to illustrate the behavior of VCGen. The general syntactic shape of the program is **let** foo = v **in** e. We start by illustrating the type inferred for foo (below), where each line is a triple of the form {pre}e {post}, showing the Hoare triple inferred for the expression e at each line.

Starting from line 10, we compute a pre-condition for the entire function with respect to a symbolic post-condition, a post-condition variable 'Post. Computing the pre-condition at each state is straightforward—we just apply the predicate transformer associated with the expression to the post-condition of the triple. For the update function, we just apply UpdTX and compute the pre-condition ϕ_1 at line 8. When traversing a let-binding, we close the post-condition being "pushed through" by λ-binding the let-bound name (as at lines 6, 4, and 2).

Deriving a type for a function

```
1  let foo this args : DST dyn ((fun args this 'Post ⇒ ϕ4) args this) =
2  {ϕ4 =SelTX args "0" (fun x ⇒ ϕ3)}
3    let x = select args "0" in
4  {ϕ3 =SelTX x "f" (fun y ⇒ ϕ2)}
5    let y = select x "f" in
6  {ϕ2 =(fun z ⇒ ϕ1) (y + 1)}
7    let z = plus y (Int 1) in
8  {ϕ1=UpdTX this "g" z 'Post}
9    update this "g" z {'Post}
10 {'Post}
```

Proceeding in this manner, we reach the top of the function body, having inferred the pre-condition ϕ_4. To give a type to foo, we generalize over the post-condition variable 'Post, and infer the type shown at line 1. The type **fun** args this 'Post $\Rightarrow \phi_4$ is now a closed term precisely capturing the semantics of foo as a predicate transformer.

Continuing on through our example, we have the derivation shown below (eliding some parts).

A derivation including a function application

```
1  let foo this args : DST dyn (FooTX args this) = ... in
2  {ϕ7 =(fun f h ⇒ (TypeOf f)=(Function FooTX) ⟹ (fun ffun ⇒ ϕ6) f h)
3                (Fun foo)}
4  let ffun = (Fun foo) in
5  {ϕ6=UpdTX global "foo" ffun (fun _ ⇒ ϕ5)}
6  let _ = update global "foo" ffun in
7  {ϕ5 =ArgsTX (fun args _ ⇒ ϕ4)} (* for some ArgsTX *)
8  let args = ... in
9  {ϕ4 =SelTX global "foo" (fun f _ ⇒ ϕ3)}
10 let f = select global "foo" in
11 {ϕ3 = Unfun(TypeOf f) args global (fun _ ⇒ ϕ2)}
12 let _ = apply f global args in
13 {ϕ2 =(fun j h ⇒ (TypeOf j)=int ⟹ (fun i ⇒ ϕ1) j h) (Int 17)}
14 let i = (Int 17) in
15 {ϕ1=UpdTX global "foo" i 'Post}
16   update global "foo" i
17 {'Post}
```

We start at the bottom with a symbolic post-condition 'Post. The basic structure of the derivation is as before—we call out two interesting elements. Consider the triples at lines 2–5 and at lines 13–15. Both of these triples have the same structure, and show how we handle the refinement formulas introduced by the constructors of the dyn type (for that matter, VCGen handles the refinements of all pure functions and constructors uniformly). Consider ffun at line 2: it has the type d:dyn{(TypeOf d)=Function FooTX}, where FooTX is the predicate transformer inferred for foo. The post-condition at that point is a predicate on ffun (i.e., **fun** ffun $\Rightarrow \phi_6$). The pre-condition we infer guards ϕ_6 with an implication allowing us to assume the refinement formula (TypeOf ffun)=Function FooTX when proving the post-condition. In a similar way, at line 13, we may assume that (TypeOf i)=int when proving the goal ϕ_1. Thus, the refinements of dyn prove useful in discharging a proof obligation.

The other interesting element is the triple at lines 11–13. Structurally, this triple is no different than any of the others—we simply apply the predicate transformer for apply. But, notice that the inferred pre-condition ϕ_3 itself contains a function over a predicate transformer—getting Z3 to handle such verification conditions requires some ingenuity, which we describe shortly (§5).

Having generated the pre-condition ϕ_7 for the whole program, a predicate (of kind heap $\Rightarrow E$) on the initial heap, we first instantiate the symbolic post-condition 'Post to the trivial predi-

cate **fun** _ _ ⇒ True, i.e., we obtain $\phi = \phi_7[(\textbf{fun}\ \_\ \_ \Rightarrow \mathsf{True})/\text{'Post}]$, which is also a predicate of kind heap ⇒ E. Next, we apply ϕ to a variable h0, bound in the initial typing context Γ_0, which includes the definitions and assumptions in the JSPrims library, e.g., the definition of the dyn type and the assumptions about Unfun, etc. Γ_0 also includes assumptions about the initial heap, e.g., that it contains the global object—call the specification of the initial heap InitialHeap :: heap ⇒ E. In §6 we show a detailed definition of the InitialHeap predicate, including a specification of the DOM API, which is accessible via the document field of the global object. Our verification goal then is to show that in the context Γ_0 the formula ϕ h0 is derivable, or, equivalently, that the context Γ_0 extended with the assumption $\neg(\phi\ \mathsf{h0})$ is unsatisfiable. More abstractly, given an MLjs program e, we aim to derive the triple {InitialHeap}e {fun _ _ ⇒ True}—if we succeed, we have proven that e executes without failing assertions.

5. Solving verification conditions

Given an initial context Γ_0, our goal is to prove (automatically) that $\Gamma_0, \neg(\phi\ \mathsf{h0})$ is unsatisfiable. This problem is well-understood when Γ_0 and ϕ h0 are first-order formulas, e.g., 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 MLjs programs are not always first-order formulas. While some experimental automated solvers for certain higher-order logics are being developed (e.g., Mona (Henriksen et al. 1995)), these remain inefficient, and provide little support for reasoning about the theories we need for MLjs, e.g., functional arrays, integers, etc. In this section, we show how to encode the proof obligations for MLjs programs in a first-order theory amenable to automated proving via Z3.

5.1 Incremental compilation of higher-order queries

We have seen two examples of higher-order formulas so far. The first case arose when verifying the JSPrims library, in particular in the implementation of apply in §3.3. There, in the first branch of the **match** (at line 8), we had to prove that a context $\Gamma_1 = \Gamma_0, \mathsf{f{:}dyn}, \mathsf{args{:}dyn}, \mathsf{this{:}dyn}, \mathsf{h{:}heap}, (\mathsf{f{=}Undef} \vee \ldots \vee \mathsf{f{=}Obj\ \_}), \mathsf{Unfun(TypeOf\ f)\ args\ this\ 'Post\ h}$ is unsatisfiable. A similar problem arises when verifying line 9 of apply, except, there, we have to prove the unsatisfiability of $\Gamma_2 = \Gamma_0, \ldots, \mathsf{f{=}Fun\ 'Tx\ fn}, \mathsf{TypeOf\ (Fun\ 'Tx\ fn){=}Function\ 'Tx}, \mathsf{Unfun(TypeOf\ f)\ args\ this\ 'Post\ h}, \neg\mathsf{'Tx\ args\ this\ 'Post\ h}$. The other case is when verifying the program of Figure 1, where our goal is (roughly) to show $\Gamma_3 = \Gamma_0, \neg((\mathsf{TypeOf\ f = Function\ FooTX}) \implies \mathsf{Unfun\ (TypeOf\ f)\ args\ global}\ (\textbf{fun}\ \_ \Rightarrow \phi_2))$ is 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 problem Γ_3 to illustrate—the same strategy applies to Γ_1 and Γ_2.

Rather than present Γ_3 directly to Z3, our query compiler implements a simple, incremental, first-order solving strategy outside of Z3. There are five steps in our query compiler.

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 Γ_3 into $\Gamma_3' = \Gamma_0, ((\mathsf{TypeOf\ f}) = \mathsf{Function\ FooTx})$, leaving a *residue* $\neg(\mathsf{Unfun\ (TypeOf\ f)\ args\ global}\ (\textbf{fun}\ \_ \Rightarrow \phi_2))$, a higher-order formula to be compiled later. The theory Γ_3' is first-order (provided Γ_0 is, which it is). The goal is to find a way to evaluate the Unfun (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 §5.2. Given the model M, we ask Z3 to evaluate the guard of the residue—in our example, this is (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 (Unfun (TypeOf f))=Tx is true in all models, i.e., we ask Z3 to prove the unsatisfiability of $\Gamma_3', \neg((\mathsf{Unfun\ (TypeOf\ f)){=}Tx})$. 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=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 $\neg\mathsf{FooTX\ args\ global}\ (\textbf{fun}\ \_ \Rightarrow \phi_2)$ is unsatisfiable in the theory Γ_3'. But, FooTX is a concrete transformer **fun** args this 'Post ⇒ ϕ_4, and our goal now contains a redex which can be reduced via several β-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. So, for many interesting programs, our query compiler may diverge, because Z3 may diverge. However, one may also be concerned that even on an effectively propositional theory (for which Z3 is complete), our query compiler introduces non-termination. While we do not prove formally that our query compiler converges, we implement a unification based heuristic to detect potential non-termination of the query compiler. We leave a formal proof of the correctness of this heuristic to future work.

5.2 Model generation and theories with quantifiers

Verifying typical MLjs programs requires reasoning about complex data structures, in particular, the DOM. The DOM is a tree-shaped data structure and providing the specification for the DOM (as we will see shortly) involves the use of an inductive predicate, which in turn requires the use of quantifiers. Since our verification procedure requires Z3 to produce models in order to resolve predicate transformers, quantifiers can be a problem—Z3's ability to find partial models for problems involving quantifiers is limited.

The listing overleaf shows a partial definition of our InitialHeap predicate (line 13), including a small fragment of our DOM specification. It says that the initial heap contains an object elt corresponding to the JavaScript access path `document.body`, and that elt satisfies the predicate EltTyping h0 elt, meaning that elt is a DOM element in heap h0.

The predicate EltTyping, defined at line 5, states that elt is an object, that 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 IsElt_trans states that IsElt is transitive in its heap argument (if the relevant fields of the element elt have not changed), and IsElt_typ expands IsElt back into EltTyping. To control how Z3 uses these quantified assumptions, we provide patterns, which serve to guide E-matching, Z3's quantifier instantiation algorithm.

Specifying the DOM (partial) using inductive predicates

```
1  type ObjField h o f = HasField h o f && TypeOf (Select h o f)=object
2                        && InDom h (Select h o f)
3
4  type IsElt :: heap ⇒ dyn ⇒ E
5  type EltTyping h elt =
6   TypeOf elt = object && ... && HasField h elt "text" &&
7   TypeOf (Select h elt "text")=string &&
8   HasField h elt "getFirstChild" &&
9   TypeOf (Select h elt "getFirstChild") =
10  Function (fun args this 'Post h' ⇒ IsElt h this &&
11        ∀child. child=Undef || IsElt h' child ⟹ 'Post child h')
12
13 type InitialHeap h0 = TypeOf global = object && ... &&
14  ObjField h0 global "document" &&
15  ObjField h0 (Select h0 global "document") "body" &&
16  EltTyping h0 (Select h0 (Select h0 global "document") "body")
17
18 assume IsElt_trans:∀ h1 h2 x.{:pattern (IsElt h1 x); (SelObj h2 x)}
19    (IsElt h1 x &&
20    (Select h1 x "text")=(Select h2 x "text") && ...
21    (Select h1 x "getFirstChild")=(Select h2 x "getFirstChild"))
22    ⟹ IsElt h2 x
23 assume IsElt_typ:∀ h x.{:pattern (IsElt h x)} IsElt h x ⇒ EltTyping h x
```

Now, consider the following JavaScript program and its MLjs counterpart. Proving the function call at line 6 requires constructing a model of the heap at that point and extracting a predicate transformer for select child "getFirstChild". This requires getting Z3 to instantiate the quantified assumptions, and to reason that since IsElt h' child (for the heap h' at that point), that child has a "getFirstChild" field, and to use EltTyping to return the expected predicate transformer as a candidate (i.e., step 2 in our query compiler), and then to confirm this candidate in step 3.

A DOM-manipulating JavaScript program and its translation

```
1 var child = document.body.getFirstChild();
2 if (child !== undefined) { child.getFirstChild() }
```

```
1 let body = select (select global "document") "body" in
2 update global "child"
3         (apply (select body "getFirstChild") body emptyObj);
4 if (select global "child" <> Undef)
5 then let child = select global "child" in
6       apply (select child "getFirstChild") child emptyObj
7 else ...
```

For simple programs such as this one, the verification proceeds without a hitch. However, as programs get more complex, Z3 has trouble producing a model in a reasonable amount of time (on the order of tens of minutes for examples a dozen lines long). Noting this kind of performance, we feared our approach was stillborn.

However, all is not lost: the expensive step in our query compilation procedure is, by far, 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 `unsat` 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 step 2, but good guesses provide us with a fast path through query compilation.

The question then becomes how to produce good guesses. Recall that we require a witness for the predicate transformer at each function call in an JavaScript program. If an efficient (but, potentially unsound) pointer analysis of JavaScript source programs can produce a may-alias set for the function call, this information can be transmitted through the JS2ML translation.

We have implemented a very simple version of such an analysis and seen some success—for JavaScript source expressions that appear syntactically to be a call to a DOM function (like `getFirstChild`), our translation emits a bit of metadata (just the name of the function). This is used by our query compiler to make a good guess for a candidate type, bypassing step 2 and making verification tractable again. With a more sophisticated pointer analysis, our translation could convey many more such guesses, ideally bypassing step 2 in the majority of cases and improving verification times substantially. Next we discuss this and other aspects of our translation, and the verification of several examples.

6. Implementation and evaluation

This section presents an empirical evaluation of our work. We apply our tool chain to verify the runtime safety of JavaScript web-browser extensions. All the major web browsers provide support for JavaScript extensions, which can 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, with one major caveat: as mentioned in the Introduction, we assume a sequential execution model for JavaScript, rather than the asynchronous model that these extensions employ in practice. A lesser caveat is that we provide extensions with an iterator-based interface to work with collections of objects in a safe manner. This differs from the standard DOM API which provides collections of objects as arrays encoded as dictionaries. We expect to handle the array idioms directly in the future.

We describe the verification task for each of our examples in detail. But, first, we briefly describe some salient points of our JS2ML implementation.

6.1 Translating JavaScript to ML

The λJS approach to specifying the semantics of JavaScript is attractive. By desugaring JavaScript's non-standard constructs into the familiar constructs of a Scheme-like programming language, λJS makes it possible to apply many kinds of standard program analyses to JavaScript programs. However, after a summer spent by the second author working with the λJS tool, and trying to get it to emit well-typed ML code, we decided to implement a tool resembling λJS, i.e., JS2ML, from scratch. Our main difficulty with λJS was its brittle front-end: parsing JavaScript is not easy (mainly because of its unusual semi-colon and off-side rule), and λJS failed to successfully parse many of the programs we were interested in.

Following the main idea of λJS, we implemented JS2ML in a mixture of C# and F#, utilizing a front-end for parsing and

desugaring JavaScript provided as part of Gatekeeper (Guarnieri and Livshits 2009), a JavaScript analysis framework. Gatekeeper provides many facilities for JavaScript analysis which we have yet to exploit, e.g., it implements a Datalog-based pointer analysis for JavaScript, which we anticipate could be used to generate the hints to our query compiler described previously.

Aside from the front-end, and the hints inserted by JS2ML, we describe the principal differences between JS2ML and λJS below.

−The output of JS2ML is typed whereas the output of λJS is untyped. JS2ML inserts the constructors of dyn such as Int and Fun throughout the translation. JS2ML translates select, update, and function application in JavaScript programs to calls of the corresponding JSPrims APIs. In contrast, λJS relies on primitive dynamically-typed operations on references and functions in Scheme.

−JS2ML translates locals to allocated objects on the heap whereas λJS locals are not objects. Our object-based locals makes our heap model more uniform.

−JS2ML has a simple annotation language for stating loop invariants. However, the support for annotations is very primitive at present—we plan to improve on this in the future.

−Guha et al. check the accuracy of their tool against a large corpus of benchmarks, showing the output of programs under the λJS semantics was identical to the original JavaScript. We have not yet tested JS2ML at a similar scale, although we plan to in the future.

6.2 Example 1: HoverMagnifier

Our first extension is HoverMagnifier, an accessibility extension: it magnifies the text under the cursor, presumably to assist a visually impaired user. 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 about 90 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`.

A fragment of HoverMagnifier

```
1 function magnify(evt) { ... }
2 var elts = document.getElementsByTagName("body");
3 var body = elts.Next();
4 if (body !== undefined)
5 {
6   body.onmousemove = function (evt){magnify(evt);};
7   body.onmousemove(dummyEv) //added for verif. harness
8 }
```

Setting up a verification harness for such a program involves two main elements. First, as mentioned earlier, we consider closed programs under a sequential execution model. So, 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 infer a weakest pre-condition for `magnify`, but since no call to it appears in the program, the pre-condition would be trivially satisfied.

More substantially, we have to provide specifications for all the APIs used by the program. For our extensions, this API is principally the DOM. We have already seen a small fragment of our DOM specification in §5.2—we elaborate on that specification here. 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 example, for `element`, we have already seen the predicate EltTyping h elt in §5.2, meaning that elt is an element in heap h.

The display below 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 this 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 `o` as the `this` parameter: `o.f = document.getElementsByTagName; o.f()`. 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 a this parameter equal to the document object doc.

The InitialHeap predicate, with a type for the `document` object

```
1  type Enumerable 'P h d = TypeOf d = object && InDom h d &&
2    HasField h d "Next" && (TypeOf (Select h d "Next")) =
3   (Function (fun args this 'Post h' ⇒ this = d &&
4                ∀(x:dyn). (x=Undef || 'P h' x) ⇒ 'Post x h'))
5
6  type DocTyping h doc = HasField h doc "getElementsByTagName"
7   && ... && (TypeOf (Select h doc "getElementsByTagName")) =
8   (Function (fun args this 'Post h1 ⇒
9    (this = doc && SingletonString h1 args &&
10   (∀ x. Enumerable IsElt h1 x ⇒ 'Post x h1))))
11
12 val global : dyn
13 type InitialHeap h0 = TypeOf global = object && ... &&
14  ObjField h0 global "document" &&
15  DocTyping h (Select h global "document")
```

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 predicate Enumerable 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 Undef (if the collection is exhausted), or returns an 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 §5.2) to include the assumption about the `document` object.

6.3 Example 2: Facepalm

Our next example is Facepalm, an extension that helps build a user's address book by automatically recording the contact information of a user's friends as they browse Facebook. It is implemented in 87 lines of JavaScript. Overall, verifying the entire extension took about 70 minutes. The verification time was dominated by the time spent in Z3. Our query compiler asked about 1,300 Z3 queries, and requires producing models to resolve function calls 95 times. The latter 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.

The main function of Facepalm is shown overleaf (`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 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 of integers (`path`) 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.

The main function and a DOM traversal function in Facepalm

```
1  function getPath(root, p) {
2      var cur=root; var path=p;
3      // needs a loop invariant
4      while(path !== undefined &&
5            cur !== undefined) {
6         cur = cur.getChild(path.hd); //needs a hint
7         path = path.tl;
8      }
9      return cur;
10 }
11 function cons(a, rest) { return {hd:a; tl:rest}; }
12 function getWebsite(elt) { ...
13   var path = cons(1,cons(0,cons(0,cons(0,undefined))));
14   return getPath(elt, path);
15 }
16 function start() {
17     var friendName, href;
18     if (document.domain === 'facebook.com') {
19         friendName = findName();
20         href = findWebsite();
21         if (href) {
22             console.log("Website on " + href);
23             console.log("Name is " + friendName);
24             saveWebsite(friendName, href);
25 }}}
```

To verify this code, the programmer needs to supply a loop invariant. Also, to make verification reasonably fast, our query compiler needs a hint at line 6, which is easily provided automatically by our front end. At the moment, JS2ML has only primitive support for annotating JavaScript source with loop invariants and other specifications. So, we describe the verification at the MLjs 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 below 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 Undef. 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.

JSPrims combinators for while-loops and for getting the heap

```
1  val _while: 'Inv :: heap ⇒ E
2     →'TxGuard::(bool ⇒ heap ⇒ E) ⇒ heap ⇒ E
3     →'TxBody::(dyn ⇒ heap ⇒ E) ⇒ heap ⇒ E
4     →guard:(unit →DST bool 'TxGuard)
5     →body:(unit →DST dyn 'TxBody)
6     →DST dyn (fun 'Post h ⇒
7        'Inv h &&                          (* Inv is initial *)
8        (∀ h1. 'Inv h1                     (* Inv is inductive *)
9        ⟹ ('TxGuard
10             (fun v h2 ⇒ (v=true ⟹ 'TxBody (fun _ ⇒ 'Inv) h2)
11                  && (v=false ⟹ 'Inv h2))
12           h1)) &&
13       (∀ h1. 'Inv h1 ⟹ 'Post Undef h1)) (* Inv implies Post *)
14
15 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[3] of `getPath` to F* 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 argument to _while (at line 7) are shown in blue. These are the predicate arguments—the first argument Inv locals h0 is provided by the programmer; the next two are wildcards (_) 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 MLjs

```
1  let getPath this args =
2   let locals = allocObject () in
3   update locals "cur" (select args "0");
4   update locals "path" (select args "1");
5
6   let h0 = get () in
7   let _ = _while (Inv locals h0) _ _
8     (fun () →(not ((select locals "path") = Undef)) &&
9              (not ((select locals "cur") = Undef)))
10    (fun () →
11      let params = allocObject () in
12      let getChild = select (select locals "cur") "getChild" in
13      let hd = select (select locals "path") "hd" in
14      update params "0" hd;
15      update locals "cur" (apply getChild (select locals "cur") params);
16      update locals "path" (select (select locals "path") "tl")) in
17  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 either be Undef, or must contain an integer hd field and also a tl field, while the `"cur"` local must contain a DOM element (or be Undef). We can state just this using the loop invariant shown in the next display.

[3] We clean up the translated code a little, using better names for variables, and removing two statements that are essentially no-ops

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 l 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 §5.2. The invariant itself Inv 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

```
1  (* Typing polymorphic lists *)
2  type IsObject h o = TypeOf o=object && InDom h o
3  type IsList :: E ⇒ heap ⇒ dyn ⇒ E
4  type ListTyping 'a h l =
5      (l=Undef ||
6      (IsObject h l &&
7        HasField h l "hd" && (TypeOf (Select h l "hd"))='a &&
8        HasField h l "tl" && IsList 'a h (Select h l "tl")))
9  assume IsList_Typing1:∀ 'a h d.{:pattern (IsList 'a h d)}
10           IsList 'a h d ⟺ ListTyping 'a h d
11 assume IsList_trans:∀ 'a h1 h2 d.{:pattern (SelObj h2 d); (IsList h1 d)}
12          (IsList 'a h1 d &&
13          (Select h1 d "hd")=(Select h2 d "hd") &&
14          (Select h1 d "tl")=(Select h2 d "tl")) ⟹ IsList 'a h2 d
15
16 (* The loop invariant *)
17 type CutIsElt h d = IsObject h d && (IsObject h d ⟹ IsElt h d)
18 type Inv locs h0 h1 =
19   (SelObj h0 global)=(SelObj h1 global) &&
20   IsObject h1 locs &&
21   HasField h1 locs "path" &&
22   HasField h1 locs "cur" &&
23   IsList int h1 (Select h1 locs "path") &&
24   ((Select h1 locs "cur")=Undef || CutIsElt h1 (Select h1 locs "cur"))
```

The invariant states that the loop does not mutate the global object at all (necessary to verify the remainder of the program after the loop); that the locs object has a "path" and "cur" fields; that the former is a list of integers; and that the latter is either Undef or a DOM element. Stating and proving the last property required an additional hint for Z3. Instead of simply writing IsElt, we had to write CutIsElt, which provides Z3 with a lemma to prove first, i.e., 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. Our work begins to put JavaScript verification on a par with the verification of these other languages. 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 interpolants to automatically infer these invariants.

6.4 Example 3: Typograf

Our final example is Typograf, an extensions that formats text a user enters in a web form. We show a small (simplified) fragment of its code below.

Message passing with callbacks in Typograf

```
1 function captureText(elt, callback) {
2  if (elt.tagName === 'INPUT'){ callback({ text: elt.value }); }
3 }
4 function listener(request, callback) {
5  if (request.command === 'captureText') {
6    captureText(document.activeElement, callback);
7 }}
8 chromeExtensionOnRequest.addListener(listener);
```

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 reqest (line 2). At line 8, Typograf registers listener as an event handler with the Chrome extension framework, by calling the function addListener.

We verified the content script for the absence of runtime errors. This program has 28 lines of JavaScript, which we verified in less than two minutes. This involved 102 queries of Z3, including resolving 7 function calls by generating Z3 models (which, as in the other cases, dominated the verification time).

Verifying this extension requires providing a model for addListener, a third-order function—it receives a second-order function (captureText) as an argument. As with HoverMagnifier, since we do not model asynchrony, we require a sequential verification harness. Instead of writing driver code to include a call, say, to listener, we give a specification to addListener that, in effect, treats it as a function that immediately call the function it receives as an argument.

A third-order specification for the Chrome API

```
1  type ChromeTyping h chrome =
2    IsObject h chrome &&
3    HasField h chrome "addListener" &&
4    (TypeOf (Select h chrome "addListener")) =
5    (Function (fun args this 'Post h' ⇒
6      (∀ args' h''.
7        (h'' = NewObj h' args' &&
8          IsObject h'' args' &&
9          HasField h'' args' "0" &&
10         HasField h'' args' "1" &&
11         IsObject h'' (Select h'' args' "0") &&
12         HasField h'' (Select h'' args' "0") "command" &&
13         TypeOf (Select h'' args' "1") =
14          (Function (fun _ _ 'Postcb hcb ⇒ 'Postcb Undef hcb)))
15   ⟹ Unfun (TypeOf (Select h' args "0")) args' Undef 'Post h'')))
```

The display above shows our (partial) specification of the Chrome API. It states that it contains a function "addListener", which expects a function as its first argument ((Select h' args "0") on the last line, i.e., listener, in our example). The specification states that it calls listener immediately in a heap h'' that differs from the input heap in that it contains a new object args' (line 7). This arguments object args' itself, in its zeroth field, contains an object with a "command" field; and in its first field, contains another function, the callback passed to listener. The callback in this case is very simple—it is the constant Undef function—but clearly, it could be given a more elaborate specification. Our verification methodology generalizes naturally to functions of an arbitrary order.

7. Related Work

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 Henglein (1994). Following that line of work, 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 in two regards. First, our JS2ML translation makes use of a similar Scheme to ML translation (combined with λJS, which we have already discussed). 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. Of course, 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 many other systems for inferring or checking types for dynamic languages—too many that covering all of them thoroughly is impossible. We focus primarily on the prior works that rely on a mechanism similar to ours, i.e., approaches 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. None of these systems handle both higher-order functions and mutable state, as we do. Additionally, we show how to embed a dynamically typed programming language within a general-purpose dependently typed programming language. This has several benefits. In contrast to the prior work, each of which required a tricky, custom soundness proof (and in the case of System D, even a new proof technique), our approach conveniently rides on the mechanized soundness result for F*. Furthermore, implementing a new type system or program verifier is a lot of work. We simply reuse the implementation of F*, with little or no modification. The only new code in F* for this project was the top-level of our query compiler and a parser for Z3 models—totaling less than a 1000 lines of F#—and even this code is useful in the verification for other F* programs. Of course, the JS2ML code is new, but implementing this is considerably easier than implementing a program verifier.

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. As explained earlier, our current heap model forbids the explicit mutation of the fields of an object used as the prototype of another object. Technically, supporting this idiom is possible in our system with a richer heap model, however, automated proving for such complex idioms is likely to be hard. Indeed, at present, Gardner et al. provide completely manual, pencil and paper proofs about small, first-order JavaScript programs. In contrast, we provide a tool for automated verification of higher-order JavaScript programs with a controlled form of prototype-based inheritance. Nevertheless, a potential direction for future work is to embed a subset of Gardner et al's separation logic style within F* 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 structire. 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. Notable among these are two control-flow analyses. We have already mentioned Gatekeeper, a pointer analysis for JavaScript—our JS2ML implementation shares infrastructure with this tool. 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 Z3 can also be seen as a very precise control flow analysis. However, as we have already discussed, there is ample opportunity for our tool to be improved by consuming the results of a source-level control-flow analysis as hints to our solver.

8. Conclusions

JavaScript has a dubious reputation in the programming language community. It is extremely popular, but is also considered to have a semantics so unwieldy that sound, automated analysis is considered an extremely hard problem. Our work establishes that with the right abstractions for reasoning about higher-order dynamically typed stores, automated program verification tools are within reach. Our current implementation is an early prototype, and we look forward to building on it. The results of this paper open the door to applying a wealth of research in automated program verification techniques to JavaScript.

References

S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security*, 2010.

A. Barth, A. P. Felt, and P. Saxena. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.

G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an smt solver. In *ICFP '10*. ACM, 2010.

R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *POPL '12*, 2012.

D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media Inc., 2008.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local hoare reasoning about dom. In *PODS '08*. ACM, 2008.

P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *POPL '12*. ACM, 2012.

S. Guarnieri and B. Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security*, 2009.

A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *ECOOP'10*. Springer-Verlag, 2010.

A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, 2011.

F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

F. Henglein and J. Rehof. Safe polymorphic type inference for scheme: Translating scheme to ml. In *FPCA*, pages 192–203, 1995.

J. Henriksen, J. Jensen, M. Jrgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, Lecture Notes in Computer Science. Springer, 1995.

D. Herman and C. Flanagan. Status report: specifying javascript with ml. In *Proceedings of the 2007 workshop on Workshop on ML*, ML '07, pages 47–52. ACM, 2007.

S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS'08*, volume 5356 of *LNCS*, 2008.

J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, 2008.

C. Schlesinger and N. Swamy. Verification condition generation using the Dijkstra state monad. Technical report, Microsoft Research, 2011.

P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: Bootstrapping certified typecheckers in f* with coq. In *POPL '12*, 2012.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP '11*. ACM, 2011.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP '10*. ACM, 2010.

D. Vardoulakis and O. Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science*, 7(2:3), 2011.