Verifying Higher-order Programs with the Dijkstra 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]

{nswamy, juanchen, livshits}@microsoft.com jww@cs.berkeley.edu cschlesi@princeton.edu

Abstract

Modern programming languages, ranging from Haskell and ML, to JavaScript, C# and Java, all make extensive use of higher-order state. This paper advocates a new verification methodology for higher-order stateful programs, based on a new monad of predicate transformers called the *Dijkstra monad*.

Using the Dijkstra monad has a number of benefits. First, the monad naturally yields a weakest pre-condition calculus. Second, the computed specifications are structurally simpler in several ways, e.g., single-state post-conditions are sufficient (rather than the more complex two-state post-conditions). Finally, the monad can easily be varied to handle features like exceptions and heap invariants, while retaining the same type inference algorithm.

We implement the Dijkstra monad and its type inference algorithm for the F* programming language. Our most extensive case study evaluates the Dijkstra monad and its F* implementation by using it to verify JavaScript programs.

Specifically, we describe a tool chain that translates programs in a subset of JavaScript decorated with assertions and loop invariants to F*. Once in F*, our type inference algorithm computes verification conditions and automatically discharges their proofs using an SMT solver. We use our tools to prove that a core model of the JavaScript runtime in F* respects various invariants and that a suite of JavaScript source programs are free of runtime errors.

Categories and Subject Descriptors D.2.4 [*Software/ Program Verification*]: Validation

General Terms Verification

Keywords Predicate transformer, Hoare monad, refinement types, dynamic languages

1. Introduction

Once the preserve of languages like Haskell and ML, programming with a mixture of higher-order functions and state is now common in the mainstream. C# and Java provide first-class closures, which interact with imperative objects in subtle ways, and in languages like JavaScript, higher-order state is pervasive. When used properly, programming with these features can promote code re-use, modularity, and programmer productivity. However, the complexity of higher-order state poses serious difficulties for program verification tools. Consider the JavaScript program below.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.


```
function incf(x) {x.f = x.f + 1; return x;};
assert (incf({f:0}).f == 1); incf=0;
```

To understand this program, we turn to Guha et al. (2010), who desugar JavaScript to λJS, an untyped lambda calculus with references and key/value dictionaries. We show the desugared JavaScript program below, with some superficial modifications—notably, rather than use an untyped lambda calculus, we present the desugared program in ML, where the typing is made explicit using a standard ML variant type *dynamic* (Henglein 1994).

```
1 let incf this args = let x = select args "0" in
2                      update x "f" (plus (select x "f") (Int 1)); x in
3 update global "incf" (Fun incf);
4 let args = let x = update (allocObject()) "f" (Int 0) in
5            update (allocObject()) "0" x in
6 let res = apply (select global "incf") global args in
7 assert(select res "f" = Int 1);
8 update global "incf" (Int 0)
```

The JavaScript function `incf` is translated to the **let**-bound λ-term incf, with two arguments: this, corresponding to JavaScript's implicit `this` parameter, and args, an object containing all the (variable number of) arguments that a JavaScript function may receive. An object is a dictionary indexed by string keys, rather than containing statically known field names. In the body of incf, the `x` argument is the value stored at the key `"0"` in the args object.

At line 3, the function incf is stored in the global object (an implicit object in JavaScript) at the key `"incf"`. At line 6 we see that a function call proceeds by reading a value out of the global object (using the library function select), 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.

Now, suppose that one wished to prove that the assertion at line 7 succeeds. One needs to reason that line 6 was indeed a call to incf. But, even proving this is non-trivial, since the call occurs via a lookup into a higher-order store, in this case, the global object. Making matters harder, the store is subject to arbitrary updates, e.g., at line 8, the incf field is modified to be Int 0 instead. Thus, in a language like JavaScript, even to reason about simple function calls one needs to reason precisely about higher-order state.

1.1 SMT-based verification of higher-order stateful programs

Our main technical contribution is a new way of structuring specifications for higher-order, stateful programs, and of inferring and automatically solving verification conditions (VCs) for these programs using an SMT solver.

Specifically, we present the *Dijkstra monad*, a new variant of the Hoare state monad of Nanevski et al. (2008a). 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. We show how to encode the Dijkstra monad in F*, a dependently typed dialect of ML (Swamy et al. 2011a), and present several examples of F* pro-

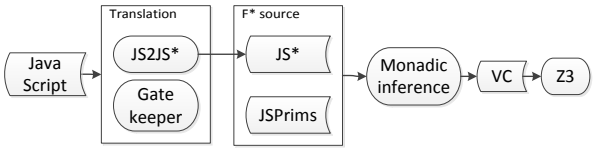
grams verified using this monad, ranging from higher-order combinators and data structures, to implementations of cryptographic protocols with stateful invariants (§2).

Verification is facilitated by a new type inference algorithm that allows stateful programs written in "direct style" to be interpreted in the Dijkstra state monad, yielding a VC for the program. We prove our inference algorithm sound, which (informally) means that if the VC computed for an F* program is provable, then the program contains no failing assertions. Despite our liberal use of predicate transformers and other higher-order logic constructs, we prove that the VCs computed by our algorithm can be represented in a first-order theory, and hence can often be discharged effectively by SMT solvers like Z3 (de Moura and Bjørner 2008) (§2.3).

To illustrate the expressiveness of our approach, we show how to program and verify implementations of abstract datatypes involving a mixture of higher-order functions and local state. Our verification technique employs a mixture of existentially quantified abstract predicates and a lightweight specification style reminiscent of separation logic, all encoded using F*'s higher-order logic. Nevertheless, we are still able to discharge verification conditions automatically using an SMT solver (§3).

1.2 An extended case study: JavaScript verification

Our most extensive case study to date involves applying the Dijkstra monad to verify JavaScript programs, a problem of growing practical importance. Our verification tool chain first translates JavaScript programs to JS* (a subset of F*), then uses our type inference algorithm to compute VCs, and, finally, solves the VCs automatically using Z3. The diagram below depicts this workflow.



Our translation, JS2JS*, adapts λJS, retargeting it to a typed language. We show how to make use of Gatekeeper (Guarnieri and Livshits 2009), an unsound but efficient pointer analysis for JavaScript, to convey verification hints in the translation, thereby improving verification times while still maintaining soundness.

Besides higher-order state, a program verifier for JavaScript programs also has to deal with dynamic typing. We show how to smoothly integrate the Dijkstra monad within a variant type dyn, a new refinement of type dynamic (Cartwright and Fagan 1991; Henglein 1994). Using this type, we program a library of runtime support for JavaScript (called JSPrims). The full version of JSPrims (available online[1]) contains approximately 1,200 lines of carefully hand-written, mechanically verified F* code, making heavy use of the Dijkstra monad and type dyn to prove that several key invariants related to the JavaScript heap are respected. Additionally, our verification guarantees that well-typed clients of JSPrims do not raise any JavaScript runtime errors. We present a simplified version of JSPrims in §4.

We report on our experience using our tools to verify a collection of JavaScript web-browser extensions for the absence of JavaScript runtime errors. We provide a verification harness that includes a partial specification of a sequential fragment of the Document Object Model (DOM)—this specification makes heavy use of the Dijkstra monad, illustrating its versatility. In each case, apart from the annotation of loop invariants, verification was automatic. The soundness of our verification results rides conveniently on the mechanized metatheory and certified implementation of F*—no JavaScript-specific extensions to the type system are required (§5).

As such, using F* as an intermediate verification language, we bring a methodology proven effective for first-order programs (using tools like Boogie (Leino and Rümmer 2010) or Why (Filliâtre and Marché 2007)), to higher-order, stateful programs.

2. The Dijkstra Monad

Monads have traditionally been used to structure effectful computations in functional programs. For example, the state monad, ST a, is the type of a computation that when evaluated in an initial heap, produces a result of type a and a final heap. In a purely functional language like Coq, ST a can be represented by the type heap → (a ∗ heap). In a language that provides primitive support for state, e.g., ML, the type ST a can just be an abstract alias for a. As a monad, the abstract type ST comes with two operations: returnST : ∀a. a → ST a and bindST: ∀a,b. ST a → (a → ST b) → ST b, and these are expected to satisfy certain laws.

To verify stateful programs, Nanevski et al. (2008b) propose Hoare Type Theory (HTT), where the main typing construct is a state monad augmented with pre- and post-conditions. The resulting monad, the *Hoare monad*, written HST pre a post, can be understood as the type h:heap{pre h} → (x:a ∗ h':heap{post x h'})[2] That is, HST pre a post is the type of a computation which when run in an input heap h satisfying the predicate pre h, produces a result x:a and an output heap h' satisfying post x h'. The HST monad comes with two operations, returnHST: ∀a, p. a → HST (p x) a p, which allows a pure value to be treated as a computation; and bindHST: ∀p a q b r. HST p a q → (x:a → HST (q x) b r) → HST p b r, which allows two computations to be composed, so long as the post-condition of the first matches the pre-condition of the second.

While the Hoare monad has proven effective in verifying stateful code in a functional language, it has some shortcomings:

−Usually, with the Hoare monad one writes specifications in the form of valid triples. This style does not lead directly to a VC generation algorithm. In order to extract a VC generation algorithm from specifications in this style, Nanevski et al. have adopted a variety of strategies in several papers. The original presentation of HTT (Nanevski et al. 2008b) employs a custom-built bidirectional typing algorithm to exploit the specific structure of HST to infer VCs. Subsequent works (Nanevski et al. 2008a) show how to embed HTT in Coq, where using a slightly different style of specification, Coq's type inference algorithm can be used to compute VCs for programs in the HST monad. However, the VCs thus computed involve a proliferation of existential quantifiers to state properties of intermediate program states. Reasoning with these quantifiers, particularly using automated SMT solvers, can be problematic.

−Additionally, writing precise specifications with the Hoare monad requires post-conditions that circumscribe the write effects of a computation, e.g., to say that the input and output heaps are the same on some domain. To achieve this, one usually requires two-state (a.k.a binary) post-conditions, or designs patterns like separation logic, adding complexity to the system.

Our key observation is that writing specifications for programs using a monad of predicate transformers (Dijkstra 1975) can alleviate the above difficulties. We call the monad the *Dijkstra monad*,

[1] Supplementary material associated with this paper is available from `http://research.microsoft.com/fstar`, and includes a compiler download, several example programs, and a technical report including proofs of all the theorems in this paper.

[2] Notation: we write x:t → t' for a dependent function whose formal parameter x of type t is named and is in scope in the return type t'; and (x:t ∗ t') for a dependent pair where x names the t-typed first component and is bound in the second component t'; and x:t{φ} for a refinement of the type t to those elements x that satisfy the proposition φ.

and write it DST a wp. This is the type of a stateful computation producing a result x:a, and whose behavior is described by the weakest pre-condition predicate transformer, wp. In a purely functional language, one may view DST a wp as an abbreviation for the type $\forall$p. h:heap{wp p h} $\rightarrow$(x:a $*$ h':heap{p x h'}). That is, in order for the output heap h' to satisfy p x h', for any predicate p, one needs to prove wp p h of the input heap h. DST provides two operations, return and bind, with the signatures shown below:

```
return: ∀a. x:a → DST a (Λp. p x)

bind: ∀a wp1 b wp2. DST a wp1
        → (x:a → DST b (wp2 x))
        → DST b (Λp. wp1 (λx. wp2 x p))
```

To inject a value x:a into the monad, one uses return, where the weakest pre-condition is simply to prove the post-condition p for x on the input heap. To compose computations, one can use bind, where the weakest pre-condition of the result is simply computed by composing the weakest pre-conditions of each component.

2.1 Simple examples

For a flavor of the Dijkstra monad, we show a selection of small ML programs and their specifications. These examples illustrate (1) the expressive power of the DST monad, in particular, the way we describe precise effects using unary, parametric post-conditions, and, how, using parametricity over the predicate transformers themselves, one can write precise specifications for higher-order programs; (2) the way in which the DST monad provides a VC generation algorithm by construction; and, (3), how the general construction can be varied to accommodate features like heap invariants for monotonic state, as well as language features like exceptions.

Primitive operations on references. We start with types for the primitives that manipulate references, where, as usual, the type heap is viewed in the logic as a partial map from references of type **ref** t to values of type t. We model the map using McCarthy's select/update theory of functional maps (McCarthy 1962), with the usual interpretation. To reduce clutter in the notation, we assume all function types are prenex quantified with their free type variables.

```
ref : x:a → DST (ref a) (Λp.λh. ∀l. l ∉ dom h ⟹ p l (update h l x))
(!) : x:ref a → DST a (Λp.λh. p (select h x) h)
(:=): x:ref a → v:a → DST unit (Λp.λh. p () (update h x v))
```

We use the function **ref** to allocate a reference cell of type **ref** a, initializing its contents to x:a. The weakest pre-condition of **ref** states that for any post-condition p, the proposition to prove on the input heap h is $\forall$l. l $\notin$ dom h $\Longrightarrow$ p l (update h l x), i.e., **ref** x returns a new heap location l not in the input heap h, and updates h at location l with the argument x. We type the dereference !(x:**ref** a) at the type a, as usual, and give it the weakest pre-condition Λp.λh. p (select h x) h, i.e., dereference returns the contents of the input heap h at location x, and leaves the heap unchanged. The type of x := v is analogous, except we return ():unit, and the weakest pre-condition states that the input heap h is updated just at x.

Note that in each case we were able to precisely state the relation between the input and output heaps by relying on a parametric unary post-condition, instead of binary post-conditions. In general, given a Hoare triple with a binary post-condition, i.e., the type h:heap{pre h} $\rightarrow$(x:a $*$ h':heap{post2 x h h'}), one can always represent it using unary post-conditions and the Dijkstra monad as DST a (Λp.λh. pre h $\wedge$ $\forall$x h'. post2 x h h' $\Longrightarrow$ p x h').

Swapping references. Take the following ML program:

```
let swap x y = let tmp = !x in let _ = x := !y in y := tmp
```

To compute a weakest pre-condition for swap, one simply computes the weakest pre-condition for each of the three commands, which are then composed using two monadic binds. The resulting type is:

```
swap: x:ref a → y:ref b → DST unit
        (Λp.λh. p () (update (update h x (select h y)) y (select h x)))
```

In contrast, using bidirectional VC generation for HST, one begins with a user-provided post-condition which is then "pushed" back through the computation. This is usually interleaved with a forward phase that computes types for **let**-bound variables. At each **let**-binding, one must close over the bound variable with an existential quantifier. The additional quantifiers make proof obligations more cumbersome to manipulate. These difficulties are avoided with predicate transformers.

Higher-order functions. Predicate parametricity is also essential for writing specifications for higher-order programs. Take the following combinator: **let** app f x = f x. Its type is

```
app: (y:a → DST b (wp y)) → x:a → DST b (wp x)
```

By being parametric in wp, app is able to abstract over the specification of its function-typed argument. Similarly, the type of **let** twice f x = f (f x), abstracts over the wp of the argument.

```
twice: (z:a → DST a (wp z)) → x:a → DST a (Λp. wp x (λy. wp y p))
```

If we were to instead abstract over the pre- and post-condition of the function argument f (rather than its wp) we would be forced to reason about twice f using invariants only, e.g., it would be impossible to prove that twice (λx. x:=!x+1) x added 2 to x. Of course, for programs with loops, we must resort to invariants. For example, here is the type of the while-combinator.

```
while: guard:(unit → DST bool wpg)
    → body:(unit → DST unit wpb)
    → DST unit (Λp.λh0. inv h0
        ∧ ∀h1. inv h1 ⟹ wpg (λb h2. b=true ⟹ wpb (λ_. inv) h2
                                          ∧ b=false ⟹ inv h2) h1
        ∧ ∀h1. inv h1 ⟹ p () h1)
```

The type above is parametric in wpg, the specification of the guard; in wpb, the specification of the loop body; and in inv, the loop invariant. The three clauses of the wp of while require the invariant to hold on the initial heap (inv h0); for the weakest pre-condition of the body (with respect to the invariant once again) to hold if the guard returns true, and otherwise for the invariant to hold immediately; and finally, for the invariant to imply the post-condition, i.e., the DST monad makes it easy to embed the standard weakest pre-condition rule for a while-loop inside the types.

Monotonic state with two-state invariants. Some applications demand variations on the basic DST monad. For instance, our supplementary material contains an example of an authentication protocol based on digital signatures, verified using the Dijkstra monad. A feature in our model of this protocol is the use of monotonic state. In particular, we make use of a stateful log of protocol events, log : **ref** (list event). For our model to be sound, at each state update, we need to verify that no event has been removed from the log. Rather than insert an explicit assertion at every command (which would be impractically tedious), we would like a variant of the DST monad that yields VCs with the necessary checks.

Our approach makes this easy to do. We provide the monad iDST, a variant of DST with a bind that checks the invariant, as shown in the signature below.

```
bind: iDST a wp1 → (x:a → iDST b (wp2 x))
        → iDST b (Λp.λh0. wp1 (λx h1. δ h0 h1 ∧ wp2 x post h1) h0).
```

This type ensures that as the heap evolves, every pair of adjacent heaps h0, h1 are related by the predicate δ h0 h1. The particular instantiation of δ depends on the application. For our authorization protocol, we instantiate δ to λh0 h1. $\forall$x. In x (select h0 log) $\Longrightarrow$ In x (select h1 log), (where In is for list membership), to ensure that the list grows monotonically.

As we will see, our type inference algorithm is generic enough to support arbitrary bind signatures, thereby allowing VC generation to be customized for the application at hand. In contrast, most program verifiers, e.g., the tools based on the Boogie framework, bake in a specific verification condition generation strategy into the tool. For example, without explicitly sprinkling assertions between every pair of commands, it is impossible to make Dafny (Leino 2010) check a two-state heap evolution invariant.

While the signature of bind requires us to prove that successive heaps are related by δ, dually, we would like to benefit from the invariant as well. When δ is reflexive and transitive, we show that a heap and any of its descendants that may result during a program execution are related by δ. Reflecting this property, we can provide two axiomatic functions, witness, which takes a snapshot of the current heap, and recall, which guarantees that the current heap is related to any prior snapshot by δ:

```
witness: unit → iDST heap (Λp. λh. p h h)
recall: h0:heap → iDST unit (Λp. λh. δ h0 h ⟹ p () h)
```

The validity of these axioms is justified by our metatheory. Intuitively, since heap is abstract, the only heap values that are available to a program are those produced by witness. Thus, in recall, we know that h0 is a heap snapshot, and by reflexivity and transitivity, it must be related to the current heap by δ.

Combining monads. Handling language features like exceptions within the Dijkstra monad is straightforward. For example, given a monad for exceptions, we can combine it with the Dijkstra monad to yield a monad for state and exceptions, with VC generation provided by construction. We first define a type result a to capture the two kinds of results a computation may produce. Normal results are tagged with the constructor V : a → result a, and exceptional results are represented using E: exn → result a, for some type of exceptions exn. Now, combining the result type with the DST monad, we write eDST a wp, which can be understood as the type ∀p. h:heap{wp p h} → (x:result a ∗ h':heap{p x h'}).

```
val return: x:a → eDST a (Λp.p (V x))
val bind: eDST a wp1 → (x:a → eDST b (wp2 x)) → eDST b wpBindE
  where wpBindE = (Λp. wp1 (λr h1. (∀ x. r=V x ⟹ wp2 x p h1)
                                  ∧ r=E _ ⟹ p r h1))
val raise: e:exn → eDST a (Λp.p (E e))
val tryWith: (unit → eDST a wp1) → (e:exn → eDST a (wp2 e))
          → eDST a (Λp. wp1 (λr h. (∀ v. r=V v ⟹ p r h)
                                  ∧(∀ e. r=E e ⟹ wp2 e p h)))
```

2.2 Monadic F⋆

To develop a type inference algorithm and metatheory for the Dijkstra monad, we introduce monadic F⋆, a programming language with a runtime semantics similar to ML (i.e., a call-by-value, higher-order programming language, with primitive support for general recursion, state, and exceptions), but with a type system that extends ML's with dependent refinements and the Dijkstra monad for functional correctness verification using an SMT solver.

Figure 1 shows the syntax of monadic F⋆. Values include variables, n-ary data constructors, abstractions, and ascriptions. Expressions additionally include applications, let bindings, and operations on references. We exclude a fix-point form, since these can be encoded using recursive data types. The full version of the paper also includes support for raising and handling exceptions.

Types include type variables a, formulas ϕ, constants for the heap, unit and references, user-defined inductive types T, type application, refinement types, and polymorphic types. We have two kinds of dependent function types: x:t → t' is the type of a pure function, and is typically used to type data constructors only. The type of effectful functions is more generally described using x:t → M t' ϕ, a dependent function with a monadic co-domain. The

v	::=	$x \mid D\ \bar{t}\ \bar{v} \mid \lambda x{:}t.e \mid \Lambda a{::}\kappa.v \mid v{:}t$	value
e	::=	$v \mid v\ v \mid v\ t \mid$ **let** $x = e$ **in** $e \mid !v \mid v := v \mid$ **ref** v	expression
	|	**match** v **with** $D\ \bar{a}\ \bar{x} \rightarrow\ e$ **else** e	
t	::=	$a \mid \phi \mid$ heap $\mid$ unit $\mid$ **ref** $t \mid T \mid t\ t$	type
	|	$x{:}t\{\phi\} \mid \forall a{::}\kappa.t \mid x{:}t \rightarrow t \mid x{:}t \rightarrow \mathsf{M}\ t\ \phi$	
ϕ	::=	$a \mid T \mid$ true $\mid$ false $\mid u = u \mid u \in u \mid \phi \wedge \phi$	formula
	|	$\phi \vee \phi \mid \neg\phi \mid \phi \implies \phi \mid \forall x{:}t.\phi \mid \exists x{:}t.\phi$	
	|	$\forall a{::}\kappa.\phi \mid \exists a{::}\kappa.\phi \mid \lambda x{:}t.\phi \mid \phi\ u \mid \Lambda a{::}\kappa.\phi \mid \phi\ t$	
u	::=	$v \mid t \mid$ select $u\ u \mid$ update $u\ u\ u \mid$ dom $u \mid$ op $\bar{u}$	logic term
κ	::=	$\star \mid E \mid x{:}t \Rightarrow \kappa \mid \alpha{::}\kappa \Rightarrow \kappa$	kind
Γ	=	$\cdot \mid \Gamma, x{:}t \mid \Gamma, a{::}\kappa$	typ. env.
Σ	=	$\mathsf{M}{::}\kappa, \mathsf{ret} = \phi, \mathsf{bind} = \phi' \mid \Sigma, T{::}\kappa\{\overline{D{:}t}\}$	signature

Figure 1. Syntax of monadic F⋆ (partial)

choice of monad M is left as a parameter in the system, except that it must be indexed by both a result type t' and a predicate transformer ϕ. By restricting monadic types to the co-domain of functions only, we borrow an insight from Swamy et al. (2011b), who use a similar restriction to develop a type inference algorithm for monadic ML. We generalize their work to the setting of a dependently typed language. However, unlike that work which supports typing a program with respect to multiple monads, in monadic F⋆ there is only a single monad. Note, polymorphic types have non-monadic co-domains corresponding to a value restriction.

Formulas ϕ describe a standard, higher-order predicate logic over a term language u that includes the values v and types t, interpreted functions from the select/update theory, other theories like arithmetic, and uninterpreted functions of the user's choosing. The logic includes variables a, uninterpreted predicates T, constants, equality, set membership, and all the usual connectives. We also include abstraction over terms and types (the two lambda forms) and the corresponding application forms.

Types are organized into kinds, κ, with two base kinds: $\star$ is the kind of types given to computationally relevant values; and kind E is for purely specificational types. In general, formulas ϕ are types with kind E. We have dependent function kinds, both on values $x{:}t \Rightarrow \kappa$ and on types $a{::}\kappa \Rightarrow \kappa$.

Type environments Γ bind variables to types and type variables to kinds. A signature Σ defines the set of inductive types $T{::}\kappa$ and their data constructors, the kind of the monad M, and the weakest pre-condition specifications of its return and bind operators.

We generally omit explicit type applications and kind annotations on type variables. These can usually be inferred.

2.3 Typing monadic F⋆

Figure 2 presents the two central judgments in the type system of monadic F⋆. The first judgment, $\Gamma \vdash e : t$, infers a value-type t for a term e in a context Γ. For effectful expressions, we have $\Gamma \vdash e : \mathsf{M}\ t\ \phi$, which infers a type t and a weakest pre-condition ϕ for an expression e from a context Γ. In both cases, the judgments are implicitly parameterized by the signature Σ (when omitted).

Monadic signature. We define K(a) as KPost(a) ⇒ heap ⇒ E for the kind of a predicate transformer from post-conditions on a-computations to pre-conditions. The type system is parametric in the choice of KPost(a)—one may pick KPost(a) to be a ⇒ heap ⇒ E for the DST monad, or result a ⇒ heap ⇒ E for the eDST monad, etc. We expect M to have kind a::⋆ ⇒ K(a) ⇒ ⋆; Σ.ret : a::⋆ ⇒ a ⇒ K(a); and Σ.bind : a::⋆ ⇒ b::⋆ ⇒ K(a) ⇒ (a ⇒ K(b)) ⇒ K(b). Observe how the signatures of Σ.ret and Σ.bind evoke a monad "one level up", i.e., they are monadic in the *kind* K(a). We expect the following three monad laws to hold, where equality is to be interpreted

$$\frac{\Gamma \text{ ok}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \text{ ok} \quad \Sigma(D) = \forall \overline{a::\kappa}.\overline{x:t'} \to t'' \quad \forall i.\Gamma \vdash t_i :: \kappa_i[\overline{t/a}] \quad \forall i.\Gamma \vdash v_i : t_i'[\overline{t/a}][\overline{v/x}]}{\Gamma \vdash D\ \bar{t}\ \bar{v} : t''[\overline{t/a}][\overline{v/x}]} \qquad \frac{\Gamma, a::\kappa \vdash v : t}{\Gamma \vdash \Lambda a::\kappa.v : \forall \alpha::\kappa.t} \qquad \frac{\Gamma \vdash v : \forall a::\kappa.t' \quad \Gamma \vdash t :: \kappa}{\Gamma \vdash v\ t : t'[t/a]}$$

$$\frac{\Gamma, x{:}t \vdash e : \mathsf{M}\ t'\ \phi}{\Gamma \vdash \lambda x{:}t.e : x{:}t \to \mathsf{M}\ t'\ \phi} \qquad \frac{\Gamma \vdash v : t' \quad \Gamma \vdash t' <: t}{\Gamma \vdash (v : t) : t} \qquad \frac{\Gamma \vdash v : t}{\Gamma \vdash v : \mathsf{M}\ t\ (\Sigma.\mathsf{ret}\ t\ v)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{M}\ t_1\ \phi_1 \quad \Gamma, x{:}t_1 \vdash e_2 : \mathsf{M}\ t_2\ \phi_2 \quad x \notin FV(t_2)}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \mathsf{M}\ t_2\ (\Sigma.\mathsf{bind}\ t_1\ t_2\ \phi_1\ (\lambda x{:}t_1.\phi_2))}$$

$$\frac{\Gamma \vdash v_1 : x{:}t \to \mathsf{M}\ t'\ \phi \quad \Gamma \vdash v_2 : t}{\Gamma \vdash v_1\ v_2 : \mathsf{M}\ t'\ \phi[v_2/x]} \qquad \frac{\Gamma \vdash v : t_0 \quad \Gamma' = \overline{a::\kappa}, \overline{x:t'} \quad \Gamma, \Gamma' \vdash D\ \bar{a}\ \bar{x} : t_0 \quad \Gamma, \Gamma' \vdash e_1 : \mathsf{M}\ t\ \phi_1 \quad \Gamma \vdash e_2 : \mathsf{M}\ t\ \phi_2}{\Gamma \vdash \textbf{match}\ v\ \textbf{with}\ D\ \bar{a}\ \bar{x} \to e_1\ \textbf{else}\ e_2 : \mathsf{M}\ t\ (\Lambda p.\lambda h.(\forall \bar{a}\bar{x}.v = D\ \bar{a}\ \bar{x} \Longrightarrow \phi_1\ p\ h) \wedge (v \neq D\_ \Longrightarrow \phi_2\ p\ h))}$$

Figure 2. Verification condition generation for monadic F*

as equi-satisfiability.

(1) $\Sigma.\mathsf{bind}\ t\ t\ (\Sigma.\mathsf{ret}\ \mathsf{v})\ \mathsf{wp} = \mathsf{wp}\ \mathsf{v}$ *(left identity)*
(2) $\Sigma.\mathsf{bind}\ t\ t\ \mathsf{wp}\ \Sigma.\mathsf{ret} = \mathsf{wp}$ *(right identity)*
(3) $\Sigma.\mathsf{bind}\ t_2\ t_3(\Sigma.\mathsf{bind}\ t_1\ t_2\ \mathsf{wp1}\ \mathsf{wp2})\ \mathsf{wp3} = \Sigma.\mathsf{bind}\ t_1\ t_3\ \mathsf{wp1}\ (\lambda \mathsf{x}.\Sigma.\mathsf{bind}\ t_2\ t_3\ (\mathsf{wp2}\ \mathsf{x})\ \mathsf{wp3})$ *(associativity)*

It is easy to check that the laws are satisfied for the DST and eDST monads (where $\Sigma.\mathsf{ret}{=}\lambda \mathsf{x}.\Lambda \mathsf{p}.(\mathsf{V}\ \mathsf{x})$ and $\Sigma.\mathsf{bind} = \mathsf{wpBindE}$), and when the δ function is reflexive, for the iDST monad as well.

Kinding and well-formedness of environments. We write Γ ok for an environment that binds unique names at well-formed kinds and well-kinded types. We omit the judgments for well-formedness of kinds and the kinding judgment—these are inherited from F* without any change and we refer the reader to Swamy et al. (2011a) for details. Additionally, we require each inductive definition in the signature Σ to be well-formed (again, this is inherited from F*).

Value typing. The first six rules in Figure 2 compute non-monadic types for values and type applications. The first four rules are straightforward. In the fifth rule, notice how we compute specifications for functions independently of any particular post-condition. The sixth rule is a subsumption rule triggered by a programmer-supplied ascription. The subtyping relation $\Gamma \vdash t' <: t$ is inherited from F*, and is implemented by a call to an SMT solver (Swamy et al. 2011a). The main interesting case is in proving a subtyping relation between two monadic types. We have $\Gamma \vdash \mathsf{M}\ t_1\ \mathsf{wp}_1 <: \mathsf{M}\ t_2\ \mathsf{wp}_2$ when $\Gamma \vdash t_1 <: t_2$ and when $\Gamma, \mathsf{p}{::}\mathsf{KPost(t)}, \mathsf{h}{:}\mathsf{heap} \models \mathsf{wp}_2\ \mathsf{p}\ \mathsf{h} \Longrightarrow \mathsf{wp}_1\ \mathsf{p}\ \mathsf{h}$.

Expression typing. The first expression typing rule corresponds to a monadic unit—it allows a value v to be injected into the monad using $\Sigma.\mathsf{ret}$. We interpret **let**-bindings as a monadic bind—we compute the type of each component, insist that the let-bound variable does not escape in the result type (although it may appear in ϕ_2), and compose the results using $\Sigma.\mathsf{bind}$. The rule for function application is standard for a dependently typed language. To type a **match**, we type each branch and compute a predicate transformer by guarding the pre-conditions of each branch by the branch condition and closing over the pattern-bound variables. We do not show specific typing rules for operations on references; these are typed according to the signatures of **ref**, (!) and (:=) shown previously.

Soundness. We prove the soundness of the type system against the operational semantics for F*—a standard small-step, call-by-value reduction relation $(H,e) \to_\Sigma (H',e')$, which relates a pair of runtime configurations consisting of a store H (mapping locations to values) and an expression e. We prove the soundness for specific instantiations of KPost(a)—here, for $\mathsf{KPost(a)} = \mathsf{result\ a} \Rightarrow \mathsf{heap} \Rightarrow E$, i.e., for a language whose primitive effects include state and exceptions. Layering other monadic effects (e.g., probabilities, or reactivity, etc.) over the the primitive ones is also possible, and the same metatheory should carry over. In the theorem below, we write $\Sigma;\Gamma \vdash (H,e) : \mathsf{M}\ t\ \phi$ to type a runtime configuration—this extends expression typing with a typing for the store H in a standard way. We also write *asResult* v for V v and *asResult* (raise *e*) for E e.

THEOREM 1 (Soundness). *Given Σ ok, and Γ, H, e, t, and wp such that $\Sigma;\Gamma \vdash (H,e) : M\ t\ wp$; and an interpretation function asHeap from runtime stores H to the type heap; and a post-condition p such that $\Sigma;\Gamma \models wp\ p\ (asHeap\ H)$ is derivable; either: (1) e is a result r and $\Sigma;\Gamma \models p\ (asResult\ r)\ (asHeap\ H)$; or, (2) there exist H', e', wp', such that $(H,e) \to_\Sigma (H',e')$, $\Sigma;\Gamma \vdash (H',e') : M\ t\ wp'$ and $\Sigma;\Gamma \models wp'\ p\ (asHeap\ H')$.*

First-order VCs. We also prove that the VC generation does not introduce higher-kinded quantification where none exists. To state this theorem, we introduce a judgment $\vdash (\Sigma;\Gamma)$ ord1, which is true of first-order contexts, i.e., those that do not use higher-order formulas in the refinement logic like $\forall \alpha::\kappa.\phi$ where $\kappa \neq \star$. The theorem states that when computing a wp for a term in such a first-order context, and when that wp is applied to a first-order post-condition and a heap, the resulting VC can be reduced to a first-order normal form, e.g., no Λs remain—this is the essence of the judgment $\Sigma;\Gamma \vdash \phi$ ord1, below. This result is important since we aim to translate VCs to SMT solvers which do not support quantification over predicates.

THEOREM 2 (First-order VCs). *If $\Sigma;\Gamma \vdash e : M\ t\ wp$; and $\vdash (\Sigma;\Gamma)$ ord1; and $\Sigma;\Gamma \vdash p$ ord1; then $\Sigma;\Gamma, h : heap \vdash (wp\ p\ h)$ ord1.*

3. Abstract predicates and local state

While Theorem 2 (First-order VCs) ensures that programs without higher-order assertions yield VCs that are first-order, this does not rule out the use of higher-order specifications. Indeed, careful use of higher-order logic in monadic F* makes it possible to reason modularly about features like local state, while still allowing automated verification via an SMT solver. We briefly present an example for illustration.

Consider the two functions below, evens and evens'.

```
let evens () = let x, y = ref 0, ref 0 in λ(). let r = !x + !y in incr x; incr y; r
let evens' () = let x = ref 0 in λ(). let r = 2∗!x in incr x; r
```

Both these functions return closures which yield the same sequence of even numbers when they are called. To see why, we can reason informally that the closure returned by evens captures two references x and y that remain local to the closure, both references always contain the same value at the entry and exit of the closure, so !x + !y is always even. A similar line of reasoning applies to evens', although it uses only one reference.

We would like to be able to give specifications to both evens and evens' that state that the references they allocate are owned by their respective closures; that their invariants are maintained irrespective of the actions of their clients; and, finally, for the clients' view of the specifications of evens and evens' to be identical, hiding the differences in the representation of their internal data structures, e.g., the number of references they use.

This is a classic problem of information hiding and modular verification considered in many lines of work (Barnett et al. 2004; Nanevski et al. 2007; O'Hearn et al. 2004; Parkinson and Bierman 2005). Monadic F* can capture the specification style of several

of these approaches naturally, e.g., the higher-order style of existentially quantified predicates of Nanevski et al. (2007), while still retaining the ability to automatically discharge proof obligations. Our solution, sketched in the remainder of this section, also illustrates common idioms we employ to control and simplify the use of the DST monad.

Note, from here on, we use the concrete syntax of F* in our examples. This is based closely on the syntax of the previous section and notations from F# and OCaml. We also use several conveniences such as curried function types and dependent pairs that are supported by our implementation.

3.1 Notation to structure the use of the DST monad

For many kinds of programs, exposing the full generality of the Dijkstra monad in specifications can be overwhelming. For example, it would be preferable to not have to think about state, exceptions, and other unrelated concerns when using the DST monad to write specifications for a pure function. To facilitate this, we define abbreviations for several common special cases of the DST monad, each a monad in its own right.

```
type Pure t requires ensures =
     DST t (Λpost h. requires ∧ ∀x. ensures x ⟹ post x h)
type Reader t requires ensures =
     DST t (Λpost h. requires h ∧ ∀x. ensures x h ⟹ post x h)
type Writer t requires ensures m =
     DST t (Λpost h. requires h ∧
               (∀ x h'. ensures h x h' ∧ Mods m h h' ⟹ post x h'))
and Mods m h h' = ∀x. x ∈ dom h ∧ x∉ m
               ⟹ x ∈ dom h' ∧ select h x = select h' x
```

For example, we use Writer t requires ensures m to type t-computations that read the heap, whose writes are confined to the set of references m, and which may also allocate new references. Notice the definition of Mods m h h'—it states that h and h' agree on all references that exist in h, except for the ones in the set m.

To make it easy to remember the interpretation of the last three indexes of Writer, we tag them with the following identifiers, each an identity function.

```
type Requires r = r      type Ensures r = r      let Modifies m = m
```

Using these tags, a function that increments an integer reference can be given the type below, hiding the complexity of predicate transformers under a notation evocative of that used by other Hoare-style verifiers.

```
x:ref int →Writer unit (Requires λh.True)
                       (Ensures λh () h'. select h' x = select h x + 1)
                       (Modifies {x})
```

3.2 Heap permissions and fragments

Monadic F* provides the flexibility to work with many different styles of specifications. For example, we can introduce a permission model (shown below) to control the use of references. We use an abstract predicate Perm r h to indicate that the program has permission to use the reference r in heap h. The pre-condition to read, write or free a reference r requires the caller to hold the permission to r. The specification of write r v ensures that the caller retains permission on r after the heap is updated. The alloc function returns a new reference r and ensures the caller has permission to use it, while free consumes the permission of the deallocated reference.

We also use features inherited from F* to axiomatize and internalize spatial connectives (like the separating conjunction from separation logic) into monadic F*. The listing below introduces a logical value Frag whose semantics is axiomatized (using the three **assume** expressions) so that Frag h fp is a heap whose domain is precisely the set of references fp and whose values on that domain agree with h. We call Frag h fp the fp-fragment of h.

```
(* Permissions on heap references*)
type Perm :: ref a ⇒ heap ⇒ E
val read: r:ref a →Reader a (Requires (Perm r))
                            (Ensures λh v. v=select h r)
val write: r:ref a →v:a →Writer unit (Requires (Perm r))
                                     (Ensures λh () h'. Perm r h'
                                                      ∧ select h' r=v)
                                     (Modifies {r})
type Fresh s h = ∀r. r ∈ s ⟹ r ∉ dom h
val alloc: v:a →Writer (ref a) (Requires λh.True)
                               (Ensures λh r h'. Fresh{r}h ∧ r∈dom h'
                                                 ∧ select h r = v ∧ Perm r h')
                               (Modifies {})
val free: r:ref a →Writer unit (Requires (Perm r))
                               (Ensures λh r h'. r∉dom h')
                               (Modifies {r})

(* Fragments of heaps *)
logic val Frag : heap →refset →heap
assume ∀fp h x. x ∈ fp ⟹ select (Frag h r) x = select h x
assume ∀fp h x. if x ∈ fp then x ∈ dom (Frag h fp) ⟺ x ∈ dom h
                else x ∉ dom (Frag h fp)
assume ∀h1 h2 fp. (∀ x. x ∈ fp ⟹ select h1 x = select h2 x)
                  ⟹ Frag h1 fp = Frag h2 fp
type Star P Q h = ∃f1 f2. f1 ∩ f2 = ∅ ∧ P (Frag h f1) ∧ Q (Frag h fp2)
type On fp P h = P (Frag h fp)
```

Using Frag, we define the heap-predicate Star P Q to be valid on a heap h if h contains disjoint fragments fp1 and fp2 such that P and Q are valid on each of those fragments. We also define On fp P, a heap predicate valid if P holds on the fp-fragment of a heap.

3.3 Specifying and verifying evens and evens'

With the heap permission model in place, we show how to program and verify our examples, evens and evens'. The main idea structuring our solution is to package a closure with an existentially quantified invariant on its local state, where the invariant encapsulates the permissions required to use the references private to the closure (thereby ensuring that a client cannot access these references). The type t below defines this existential package:

```
type t = MkT: Inv::heap ⇒ E →fp:refset →(unit →evens_t Inv fp) →t
and evens_t Inv fp =
     Writer int (Requires (On fp Inv))
                (Ensures λh x h'. x mod 2 = 0 ∧ On fp Inv h')
                (Modifies fp)
```

A t-value MkT Inv fp f is a triple consisting of a heap-predicate Inv, a set fp of references (representing the footprint of the invariant), and a closure f:unit →evens_t Inv fp, a function in the Writer monad whose pre-condition requires that Inv be valid on the fp-fragment of the input heap, and which ensures that it returns an even integer; that Inv remains valid on the same fragment of the output heap; and that only references in fp may be modified. Observe that this type reveals no information about the structure of the local state of f, although it guarantees that f always returns an even integer. (Note, if we wanted to prove that each call to f returned the next even integer, we would have to augment MkT with additional (ghost) state. This is relatively straightforward to do, although we omit it here for simplicity.)

Next, we show the type of evens and evens' (the same for both).

```
val evens,evens': unit →Writer t
  (Requires λ h. True)
  (Ensures λ h v h'. Fresh (fp v) h ∧ On (fp v) (Inv v) h')
  (Modifies {})
```

This is the type of a function in the Writer monad with a trivial pre-condition returning a t-value. To state the post-condition, we project the footprint and the invariant component of the v:t result using the logical projectors fp:t →refset and Inv::t ⇒ heap ⇒ E that

are provided in F*'s logic for the data type t. We assert, first, that the references in the footprint (fp v) are freshly allocated, and second, that the existentially bound invariant (Inv v) is valid on the fragment of the output heap h' corresponding to the footprint. Additionally, both functions do not modify any existing reference.

The listing below shows the implementation of evens and evens', each returning the closures we showed earlier packaged as a t-value with a suitable invariant. The invariant for evens is Inv1 x y, which states that the closure holds the permission to x and y, and that the values of these references are always the same in the heap. The invariant for evens' is just Perm x, just stating that its closure holds permission to x.

```
type Inv1 x y h = Perm x h ∧ Perm y h ∧ select h x=select h y
let evens () = let x,y = alloc 0, alloc 0 in
               let f () : evens_t (Inv1 x y) {x,y} =
                 let r = read x + read y in incr x; incr y; r in
                 MkT (Inv1 x y) {x,y} f

let evens' () = let x = alloc 0 in
                let f () : evens_t (Perm x) {x} =
                  let r = read x in incr x; 2*r in
                  MkT (Perm x) {x} f
```

These examples, while particularly simple, illustrate a general pattern in monadic F* for writing specifications over local state. We make use of this pattern in programming other, more complex examples, e.g., a verified ring buffer programmed using arrays, local state, and a record of closures to mimic an object-oriented programming style. We discuss this briefly in §5.

4. Dynamic typing in monadic F*

We turn now to our main case study, JavaScript verification. While JavaScript is increasingly used for both web and server-side programming, it is a challenging language for verification and analysis, in part due to its highly dynamic nature. We implement a translation, JS2JS*, from JavaScript to F*, based closely on λJS (Guha et al. 2010). The image of this translation is JS*, an idiomatic subset of monadic F*.

A first difficulty in translating JavaScript to F* is to represent JavaScript's dynamic typing. In principle, to handle dynamic typing in an ML-like language, one simply provides a variant type dyn, with cases for each primitive type constructor in the language, e.g., Int : int → dyn, Fun: (dyn → dyn) → dyn, while systematically inserting injections and projections to and from type dyn. So, intuitively, we compose the λJS translation with a translation that adds type dyn systematically to produce a JS* program. JS* programs are linked against a library called JSPrims, which includes type dyn, as well as functions providing runtime support for coercions, object manipulation, prototype traversal, etc. The formalization of JS2JS* and the full JSPrims library are available online.

4.1 A refined type dynamic

The basic approach of inserting the standard type dyn systematically throughout the program is conceptually simple, but also not very useful—nearly every term is typed at dyn. Our first step towards verifying JavaScript programs, then, is a new *refined* type dyn that recovers the precision of static typing.

The listing below shows the (partial) definition of our type dyn, suitable for use with JavaScript programs translated to JS*. For example, we have the constructor Num, which allows a number n:float to be injected into the dyn type. However, the type of Num n is d:dyn{TypeOf d = float}, where TypeOf: a::⋆ ⇒ a ⇒ E is an uninterpreted function from values to types in the refinement logic. In effect, the refinement formula recovers the static type information. The case of strings and objects is similar—note, an object is represented as a mutable map from strings to dyn values.

```
type dyn =
  | Num : float → d:dyn{TypeOf d = float}
  | Str : string → d:dyn{TypeOf d = string}
  | Obj : ref (map string dyn) → d:dyn{TypeOf d = object}
  | Fun : ∀wp::dyn ⇒ dyn ⇒ (dyn ⇒ heap ⇒ E) ⇒ heap ⇒ E.
            (this:dyn → args:dyn → DST dyn (wp args this))
          → d:dyn{TypeOf d = AsE wp}
  | Undef : d:dyn{TypeOf d = undef} ...
```

The type of Fun merits closer attention. JavaScript functions always take two arguments—the first is for the implicit `this` parameter; the second is an object containing a variable number of fields, one for each of the variable number of actual parameters. So, to a first approximation, the type of Fun is (dyn → dyn → dyn) → dyn, but this is, of course, too imprecise. To recover precision, we type each function in the DST monad and refine the type of Fun to record the predicate transformer of the function in the refinement logic. From the type d:dyn{TypeOf d = AsE wp}, we can conclude that the underlying value is a function of type this:dyn → args:dyn → DST dyn (wp args this). (Note, AsE is just a type constructor that coerces the kind of wp to E-kind.)

Our full library includes other JavaScript primitive types; a more complicated type for Fun, which allows functions to also be treated as objects (as required by JavaScript); and types/functions to model exceptions and JavaScript's control operators. We leave them out of this paper due to space constraints.

4.2 An API for JS*

Next we discuss some key fragments of JSPrims, simplified substantially for the purposes of the paper. JSPrims defines the set of JavaScript runtime errors we capture.

Safe field selection. To select a field f:string from an object o:dyn, JS* programs call selField o f (corresponding to the JavaScript source operation `o.f` or `o["f"]`). The implementation of selField (line 8 below) elides several details of prototype chain traversal, functions as objects, etc. In its most basic form, selecting a field requires o to be an object. We dereference the map location, lookup the key f and return the value, if it exists.

```
 1 define SelField h (Obj l) f = Map.select (select h l) f
 2 define HasField h (Obj l) f = In f (Map.domain (select h l))
 3 val unreachable: unit → DST a (Λp.λh. false)
 4 val lookup: m:map a b → x:a → option (y:b{y=Map.select m x})
 5 val selField: o:dyn → f:string → Reader dyn
 6     (Requires λh.TypeOf o=object∧HasField h o f)
 7     (Ensures λh v. v=SelField h o f)
 8 let selField o f = match o with
 9 | Undef | Str _ | Int _ | Fun _ → unreachable ()
10 | Obj l → match lookup !l f with Some v → v
11 | None → unreachable()
```

We aim to give a specification to selField that ensures that well-typed JS* clients avoid various JavaScript errors. Projecting a field from a non-object is prevented by the clause in the precondition (line 6) which requires TypeOf o = object. We also ensure that the field f be among the defined fields of the objects (HasField h o f).

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. At line 7, the specification ensures that selField returns the contents of the f field of o, and that the heap is unchanged.

Note, the definitions at lines 1-2 provide two functions in the logic, corresponding to projecting a field from an object and to testing whether a key exists among the fields of an object. We interpret the map type as a functional array, and use functions Map.select/Map.update/Map.domain, with the usual interpretation.

Safe field update. To update a field f:string from an object o:dyn with a value v:dyn, JS⋆ programs call updField o f v (corresponding to the JavaScript source operation `o.f = v` or `o["f"] = v`).

The specification of updField has a similar form to that of selField. The caller is required, first, to prove that o is an object (trying to update a non-object is a JavaScript error). Note, the field f need not exist in o—JavaScript permits adding fields to an object "on the fly". The specification also states that updField returns Undef and updates just the heap accordingly. The modifies clause uses a projector provided in F⋆'s logic Obj_proj_0 to state that only the reference held in o is modified.

```
 1 define UpdField h (Obj l) f v =
 2   update h l (Map.update (select h l) f v)
 3 val insert: m:map a b →x:a →y:b
 4               →m':map a b{m'=Map.update m x y}
 5 val updField: o:dyn →f:string →v:dyn →Writer dyn
 6     (Requires λh. TypeOf o=object)
 7     (Ensures λh u h'.u=Undef ∧ h'=UpdField h o f v)
 8     (Modifies {Obj_proj_0 o})
 9 let updField o f v 'Post h = match o with
10 ... | Undef | Str _ | Int _ | Fun _ →unreachable ()
11 | Obj l →l := insert !l f v; Undef
```

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(a1,...,an)`, where args is an object containing the n actual parameters. As in the other cases, our goal is to ensure that function applications 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.

```
1 val apply: f:dyn →this:dyn →args:dyn →DST dyn (Λp.λh.
2       ∃ wp. TypeOf f = AsE wp ∧ wp args this p h)
3 let apply f this args = match f with Fun _ fn →fn this args
4 ... | Undef | Str _ | Int _ | Obj _ →unreachable ()
```

The implementation of apply is straightforward—it checks that f is a function and applies it. The specification of apply requires the client to prove that f is indeed a function with some wp, and then to prove the pre-condition computed by that wp for the particular arguments and post-condition needed at the call site.

4.3 Computing VCs for JS⋆

We prove that VC generation always succeeds for any loop-free JS⋆ program. Programs with loops must be annotated with invariants. We write ⟦e⟧ for the translation of a λJS program to JS⋆.

THEOREM 3 (VC generation is complete on JS⋆). *Given a loop-free λJS term* e. *Then, for the signature* $\Sigma_{JSPrims}$, *there exists* wp *such that* $\cdot \vdash$ ⟦e⟧ : *M dyn wp*.

By default, the VC for a JS⋆ program is not first-order. Notably, each call to apply introduces an existentially bound predicate transformer. Our full paper shows how such existentially bound transformer variables can be eliminated using the model-finding features of an SMT solver. However, this procedure can be quite expensive (cf. the verification time of Facepalm in the next section).

We describe here a more efficient approach that we also implement. Specifically, instead of apply we use apply_hint shown below, where, instead of existentially quantifying over the wp, we require the caller to instantiate the wp explicitly.

```
val apply_hint: wp::dyn⇒ dyn⇒ (dyn⇒ heap⇒ E )⇒ heap⇒ E
→f:dyn →this:dyn →args:dyn
→DST dyn (fun post h ⇒ TypeOf f=AsE wp ∧ wp args this post h)
```

Unlike apply, the type of apply_hint satisfies our ord1-restriction, and the VCs produced when calling apply_hint can be handled within a first-order solver like Z3.

Of course, one still needs a way to compute the type argument of apply_hint. We use Gatekeeper, a pointer analysis for JavaScript, to produce a set of possible function call targets. We carry this information in the JS2JS⋆ translation and use it to compute the type arguments of apply_hint. The example below illustrates this procedure, where the JavaScript program on the left is translated to the JS⋆ program on the right.

```
function foo(x) {...}          | let foo this args = ... in
foo({f:0});                    | update global "foo" (Fun 'U0 foo);
foo = 17;                      | apply_hint 'U0 (select global "foo")
                               |       global (...);
                               | update global "foo" (Int 17)
```

Each lambda-term in the translated program produces an application of the Fun constructor. For each such function, 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 FooTX, 'U0 is unified with 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 apply_hint, 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 FooTX, the witness argument to apply_hint is suitably unified too. We also support variants of apply_hint in case Gatekeeper cannot resolve a call target to a singleton. Note, we do not rely on the soundness of Gatekeeper (in fact, it is occasionally unsound)—an incorrect instantiation will be trapped by the theorem prover.

5. Experimental evaluation

To date, we have verified nearly 3,000 lines of F⋆ code using the Dijkstra monad, summarized in Figure 3. The verification time (column **TC**) was collected on a 3.2GHz Windows 7 machine. The first class of examples is a set of 26 programs, listed in a single line in the table as ex0–ex25, and totaling 400 lines of code. These examples include the code discussed in §2.1, and others of a similar flavor. We have also ported three programs implemented by Chen et al. (2010) in Fine to monadic F⋆. The first, ac, is a simple access control scheme. Next, automaton is a program that implements a type-state protocol on files—coded in monadic F⋆, we no longer need to thread affine tokens and a store. Finally, iflow is a monadic encoding of information flow control. Ported to monadic F⋆, it highlights monadic layering—we can easily combine the Dijkstra state monad with a programmer-defined information flow monad. The programs auth and auth2 implement variations of a protocol that used digital signatures for authentication. They use the iDST monad to maintain an invariant on mutable memory that is used to track a collection of protocol events. The examples evens is the local state example of §3. The programs ringbuf1 and ringbuf2 program and verify a ring buffer based on a problem specification provided by the 2012 VSTTE verification competition. JSPrims is our most complex example: it provides a precise type interface suitable for verifying functional correctness of JS⋆ programs, while implementing runtime support including coercions, prototype traversal, and the JavaScript calling convention. It uses the eDST and iDST monads to include reasoning about exceptions and a two-state invariant that controls how the JavaScript heap evolves.

The remaining programs are JavaScript web-browser extensions translated to JS⋆ for verification. These extensions are based on those studied by Guha et al. (2011). We prove each extension free of runtime errors, assuming a sequential model for the DOM. Our

Name	LOC(JS)	TC (sec)	Description
ex0–ex25	406	9.1	Classic combinators and data struct.
ac	38	6.5	Access control (PLDI '10)
automaton	53	7.5	Typestate on files (PLDI '10)
iflow	115	10.1	Information flow monad (PLDI '10)
auth	52	6.2	Digital sig. with monotonic state
auth2	53	6.8	Variation of authentication above
evens	59	2.9	Local state for even streams
ringbuf-1	192	22.0	Ring buffer with global invariants
ringbuf-2	154	21.8	Ring buffer with local state
JSPrims	1,131	63.5	Runtime support for JavaScript
Untiny	59 (9)	11.0	Send selected URL
Delicious	65 (13)	11.3	Bookmark selected text
Password	111 (29)	42.7	Store and retrieve passwords
HoverMagn	60 (23)	38.1	Magnify text under the cursor
Typograf	106 (28)	65.5	Format text a user inputs
Facepalm	270 (82)	718.0	Find contacts from Facebook
Total	**2,924 (184)**	**17m 23s**	

Figure 3. Summary of experiments

verification methodology is designed for proving the functional correctness of programs, although, in the absence of specifications for JavaScript programs in the wild, our tool checks for safety.

To simplify verification, in some cases, we model collections of objects by iterators, whereas the standard DOM API provides collections of objects as arrays encoded using dictionaries. We expect to support the array idioms in the near future, although precisely modeling asynchrony in the DOM is substantial future work. The remainder of this section discusses three extensions in detail.

5.1 HoverMagnifier

Our first extension is HoverMagnifier, an accessibility extension. It magnifies the text under the cursor. A key part of its code is shown below—it involves the manipulation of a collection of DOM elements.

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

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 5 sets an event handler, `magnify`, to be called whenever the user's mouse moves—we elide the definition of `magnify`.

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 6 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 pre-condition would be trivially satisfied. (In §5.3, we show how one can avoid writing such driver code by instead writing better specifications for library functions.)

We also 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.

```
1  type EltTyping h elt =
2   TypeOf elt = object ∧ ... ∧ HasField h elt "text"
3   ∧ TypeOf (SelField h elt "text")=string
4   ∧ HasField h elt "getFirstChild"
5   ∧ TypeOf (SelField h elt "getFirstChild") =
6       AsE (λargs this post h'. IsElt h this ∧
7            ∀child. (child=Undef ∨ IsElt h' child) ⟹ post child h')
8  and IsElt :: heap ⇒ dyn ⇒ E
9  assume IsElt_trans:∀ h1 h2 x.
10   (IsElt h1 x ∧
11   (SelField h1 x "text")=(SelField h2 x "text") ∧ ...
12   (SelField h1 x "getFirstChild")=
13     (SelField h2 x "getFirstChild"))
14   ⟹ IsElt h2 x
15 assume IsElt_typ:∀ h x. IsElt h x ⟹ EltTyping h x
```

The predicate EltTyping (line 1) states that elt is an object, it has (among others) a field `"getFirstChild"`, and that this field is a function whose specification is given by the predicate transformer at lines 6–7. 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 Undef). Capturing this specification involves the use of an abstract inductive predicate IsElt, and then providing two assumptions (at the bottom of the display) giving it an interpretation. 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.

The next listing shows the predicate DocTyping, a partial specification for the `document` object (line 1). It states that the object doc contains a field `"getElementsByTagName"` that stores a function-typed value. The pre-condition for this function requires that it be called with its this argument set to the doc object itself. 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.

```
1 type DocTyping h doc = HasField h doc "getElementsByTagName"
2  ∧ ... ∧ TypeOf (SelField h doc "getElementsByTagName") =
3  AsE (λargs this post h1.
4   (this = doc ∧ SingletonString h1 args ∧
5   (∀ x. Enum IsElt h1 x ⟹ post x h1)))
6 and Enum p h d = TypeOf d=object ∧ d ∈ dom h ∧
7   HasField h d "Next" ∧ TypeOf (SelField h d "Next") =
8   AsE (λargs this post h'. this = d ∧
9     ∀x. (x=Undef ∨ p h' x) ⟹ post x h')
```

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 5. It states that the function does not change the heap, and that the object x returned satisfies the predicate Enum IsElt h1 x.

The predicate Enum is shown at line 6. 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 a value satisfying the predicate p h' x. As with other functions, `"Next"` expects its this pointer to be the enclosing collection.

Finally, to connect these specifications of the DOM to the program itself, we type the program in an initial heap h0 satisfying the InitialHeap h0 predicate overleaf, which states that a document object is reachable from the global object.

```
val global : dyn
type InitialHeap h0 = TypeOf global = object ∧ ...
 ∧ HasField h0 global "document"
 ∧ TypeOf (SelField h0 global "document")=object
 ∧ DocTyping h0 (SelField h0 global "document")
```

5.2 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. The verification time of Facepalm 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 for the DOM when configuring Gatekeeper to help.

```
 1 function getPath(root, p) {
 2     var cur=root; var path=p;
 3     while(path !== undefined ∧ //needs loop invariant
 4           cur !== undefined) {
 5        cur = cur.getChild(path.hd); //needs a hint
 6        path = path.tl; }
 7     return cur; }
 8 function start() {
 9     var friendName, href;
10     if (document.domain === 'facebook.com') {
11         friendName = findName();
12         href = findWebsite();
13         if (href) {
14             console.log("Website on " + href);
15             console.log("Name is " + friendName);
16             saveWebsite(friendName, href); }}}
```

The main function of Facepalm is shown above (`start` at line 8). At a high level, this extension checks to see if the page currently being viewed is a Facebook page (line 10). 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 11).

A second traversal finds the friend's contact and website information (line 12). 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 16).

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. The next listing shows the translation (slightly cleaned up) of `getPath` to JS* for verification, starting with the signature of a function `get` which returns the current heap (useful for stating invariants).

At lines 3–5, 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 from §2.1. The first three arguments to while (at line 8) 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 F*. The fourth argument is the thunk representing the loop guard, and the last argument is a thunk for the loop body.

```
 1 val get: unit → DST heap (λp h. p h h)
 2 let getPath this args =
 3   let locals = allocObject () in
 4   updField locals "cur" (selField args "0");
 5   updField locals "path" (selField args "1");
 6
 7   let h0 = get () in
 8   let _ = while (Inv locals h0) _ _
 9     (λ_ _. (not ((selField locals "path") = Undef)) ∧
10               (not ((selField locals "cur") = Undef)))
11     (λ_ _.
12       let ps = allocObject () in
13       let getChild = selField (selField locals "cur") "getChild" in
14       let hd = selField (selField locals "path") "hd" in
15       updField ps "0" hd;
16       updField locals "cur" (apply getChild (selField locals "cur") ps);
17       updField locals "path" (selField (selField locals "path") "tl");
18       ( ))
19   in selField locals "cur"
```

The code of the loop guard is straightforward. The body allocates an object ps to pass arguments to the function getChild. The parameters at the call on line 16 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 below.

```
 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 (SelField h l "hd")=a ∧
 8       HasField h l "tl" ∧ IsList a h (SelField h l "tl")))
 9 assume typing1:∀ a h d.
10         IsList a h d ⟺ ListTyping a h d
11 assume trans:∀ a h1 h2 d.
12         (IsList a h1 d ∧
13         (SelField h1 d "hd")=(SelField h2 d "hd") ∧
14         (SelField h1 d "tl")=(SelField h2 d "tl")) ⟹ IsList a h2 d
15 (* The loop invariant *)
16 type CutIsElt h d = IsObject h d ∧ (IsObject h d ⟹ IsElt h d)
17 type Inv locs h0 h1 =
18   (GetFields h0 global)=(GetFields h1 global) ∧
19   IsObject h1 locs ∧
20   HasField h1 locs "path" ∧
21   HasField h1 locs "cur" ∧
22   IsList int h1 (SelField h1 locs "path") ∧
23   ((SelField h1 locs "cur")=Undef ∨
24     CutIsElt h1 (SelField h1 locs "cur"))
```

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 of IsElt. The invariant Inv itself is defined at line 17. 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 states that: (1) the loop does not mutate the global object (necessary to verify the code after the loop); (2) the locs object has fields `"path"` and `"cur"`; (3) `"path"` is a list of integers; and (4) `"cur"` is either Undef or a DOM element. Stating and proving (4) required an additional hint (CutIsElt) for Z3 to first prove that `"cur"` is an object and then that it is a DOM element.

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 interpolants to automatically infer these invariants.

5.3 Typograf

Our final example is Typograf, an extension that formats text a user enters in a form. When Typograf receives a request to capture the text, it calls `captureText`, which calls the `callback` function in the request (line 3). At line 9, Typograf registers `listener`, which calls `captureText`, as an event handler with the Chrome extension framework, by calling the function `addListener`. We verified Typograf for the absence of runtime errors. We show a simplified fragment of its code below.

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

Verifying this extension requires providing a specification for `addListener`, a third-order function—it receives a second-order function (`listener`) as an argument. As the example below shows, our verification methodology works naturally at higher order, and the same methodology can be used to write specifications for functions of an arbitrary order.

```
1  type ChromeTyping h chrome =
2    IsObject h chrome ∧
3    HasField h chrome "addListener" ∧
4    TypeOf (SelField h chrome "addListener")) =
5    AsE (λargs this post h'.
6      (∃ wp. TypeOf (SelField h' args "0") = AsE wp ∧
7      (∀ args' h''.
8        (not (InDom h' (Loc args)) ∧
9          h'' = Alloc h' args' ∧
10         IsObject h'' args' ∧ HasField h'' args' "0" ∧
11         HasField h'' args' "1" ∧
12         IsObject h'' (SelField h'' args' "0") ∧
13         HasField h'' (SelField h'' args' "0") "command" ∧
14         TypeOf (SelField h'' args' "1") =
15           AsE (λ_ _ postcb hcb. postcb Undef hcb))
16     ⟹ wp args' Undef post h'')))
```

Since we do not yet model asynchrony, we require a sequential verification harness. However, (in contrast with HoverMagnifier), we show how instead of writing driver code to include a call to `listener`, we give a specification to `addListener` that, in effect, treats it as a function that immediately calls the function it receives as an argument. Using more realistic drivers is future work.

The listing above shows our (partial) specification of the Chrome API. It states that Chrome contains an `"addListener"` function, which (at line 6) expects a function as its first argument, 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 9). 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.

6. Related work

Our verification methodology is connected to a long line of literature of Hoare logic and dependently typed programming languages. In addition to the connections already discussed, our methodology is related to the characteristic formulae of Charguéraud (2011). These formulae represent programs in higher-order logic, such that two programs are equivalent if and only if their characteristic formulae are logically equivalent. Charguéraud shows how to compute a characteristic formula and manipulate it interactively in a theorem prover. In contrast, our predicate transformers are designed to be provable using an automated first-order theorem prover.

Also related is work on F* and related languages. Borgström et al. (2009) present an application of the Hoare monad within F7, a language subsumed by F*. Borgstrom et al. (2011) use substructural state on top of a Hoare-like monad to model local state. Bhargavan et al. (2010) adopt syntactic conventions to verify higher-order programs in a language with only first-order refinement types. However, none attempts the combination of type inference, higher-orderness, and state.

Fournet et al. (2013) use our JSPrims library to prove a compiler from a subset of F* to JavaScript fully abstract. A key part of their proof methodology involves the use of the Dijkstra monad to state and prove invariants of the translation. As such, their work constitutes separate validation of the expressiveness of the Dijkstra monad and the effectiveness of our verification methodology.

Another line of work on verifying higher-order programs is via higher-order model checking (Kobayashi et al. 2011) or via liquid types (Rondon et al. 2008). These approaches aim to be automated by discovering invariants. However, these systems generally do not handle state. Combining model checking or abstract interpretation based approaches with our work is likely to pay dividends, particularly in the inference of invariants.

Our case study of JavaScript verification is related to work that equips dynamic languages with 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. They were able to statically discover certain kinds of runtime errors in Scheme programs via their translation to ML. Our JS2JS* translation makes use of a similar translation (combined with λJS). 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 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. 2012b) 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. Chugh et al. (2012a) extends System D to provide a type system for JavaScript. They rely on explicit type annotations, and do not provide type inference or soundness of the type system.

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. 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* for JavaScript verification. 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 Z3 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 JavaScript programs in order to statically find program errors. The goals are similar to our JavaScript verification, 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.

7. Conclusions

Structuring specifications for higher order, stateful programs in the style of the Dijkstra monad has a number of benefits. As we have argued, it lends itself naturally to type inference, the VCs it computes can be handled by an SMT solver, and it is flexible enough to accommodate various language disciplines (ranging from dynamic typing, local state, and higher order stores, to exceptions and monotonic state). Indeed, the basic methodology makes very few assumptions about the underlying monad, allowing our tools to be repurposed for other verification tasks—our initial experience combining the Dijkstra monad with an information flow monad is illustrative. Going further in this direction, we conjecture that adapting existing monads for functional reactivity to the Dijkstra-style is a natural way for handling asynchrony within our framework.

Using our JavaScript verification tool chain we have shown that with the right abstractions for reasoning about higher-order, dynamically typed stores, automated program verification tools are within reach for JavaScript. Despite some limitations, ours is the first tool to enable sound, precise, semi-automated, modular verification for a sizeable subset of JavaScript, including its higher-order and stateful features.

References

M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3, 2004.

K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *APLAS*, pages 47–62, 2010.

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

J. Borgström, A. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. Technical Report TR-2009-97, Microsoft Research, 2009.

J. Borgstrom, J. Chen, and N. Swamy. Verifying stateful programs with substructural state and hoare types. In *PLPV*, Jan. 2011.

R. Cartwright. *A Practical Formal Semantic Definition and Verification System for TYPED LISP*. Garland Publishing, New York, 1976.

R. Cartwright and M. Fagan. Soft typing. In *PLDI*, 1991.

A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, 2011.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*, 2010.

R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *OOPSLA*, 2012a.

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

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.

J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

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

P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *PODS*, 2008.

P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for Javascript. In *POPL*, 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*, 2010.

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

S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, pages 238–255, 2009.

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, pages 222–233, 2011.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.

K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, 2010.

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

A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *ESOP*, pages 189–204, 2007.

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

A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008b.

P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, 2004.

M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

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

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011b.

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

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