Fully Abstract Compilation to JavaScript

Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand,
Pierre-Yves Strub, Benjamin Livshits

Fully Abstract Compilation to JavaScript

Cédric Fournet Nikhil Swamy Juan Chen Pierre-Evariste Dagand Pierre-Yves Strub[1] Benjamin Livshits

Microsoft Research and MSR-INRIA[1]

{fournet, nswamy, juanchen, livshits}@microsoft.com dagand@cis.strath.ac.uk pierre-yves@strub.nu

Abstract

Many tools allow programmers to develop applications in high-level languages and deploy them in web browsers via compilation to JavaScript. While practical and widely used, these compilers are ad hoc: no guarantee is provided on their correctness for whole programs, nor their security for programs executed within arbitrary JavaScript contexts. This paper presents a compiler with such guarantees. We compile an ML-like language with higher-order functions and references to JavaScript, while preserving all source program properties. Relying on type-based invariants and applicative bisimilarity, we show full abstraction: two programs are equivalent in all source contexts if and only if their wrapped translations are equivalent in all JavaScript contexts. We evaluate our compiler on sample programs, including a series of secure libraries.

This version supercedes the version in the official proceedings of POPL '13. In particular, we fix `upfun` in Figure 4, rectifying an experimental error that rendered the previous `upfun` an insufficient protection on several popular browsers. The new version is confirmed to work on IE 9, IE 10, Chrome 23, and Firefox 16. Thanks to Karthik Bhargavan for pointing out the error.

Categories and Subject Descriptors D.2.4 [*Software/ Program Verification*]: Validation; D.3.4 [*Processors*]: Compilers; D.4.6 [*Operating Systems*]: Security and Protection—Verification.

Keywords Program equivalence; full abstraction; refinement types.

1. Introduction

Many tools allow programmers to develop applications in high-level languages and deploy them in web browsers via compilation to JavaScript. These include industrial compilers like GWT for Java, WebSharper and Pit for F#, and Dart, as well as several academic efforts like Links (Cooper et al. 2006) and Hop (Serrano et al. 2006). While practical and, in some cases, widely used, these compilers are *ad hoc*: no guarantee is provided on their correctness for whole programs, nor their security for programs executed within arbitrary JavaScript contexts.

The lack of security against JavaScript contexts is of particular concern, since compiled code is routinely linked with libraries authored directly in JavaScript. Libraries like jQuery and Prototype are widely used, provide improved support for several core web-programming tasks, but do so by making use of highly dynamic features of JavaScript, e.g., by redefining properties of predefined objects. Less well-known libraries are also routinely included in pages, often by simply including a pointer to the code served from a potentially untrustworthy URL. It is also common practice to include rich third-party content (e.g., advertisement scripts) in the same context as trusted JavaScript code. In all those cases, linking with a malicious or buggy script can easily break invariants of compiled code, compromise its security, and, in general, render any reasoning principles of the source programming language inapplicable to the JavaScript code that is actually executed.

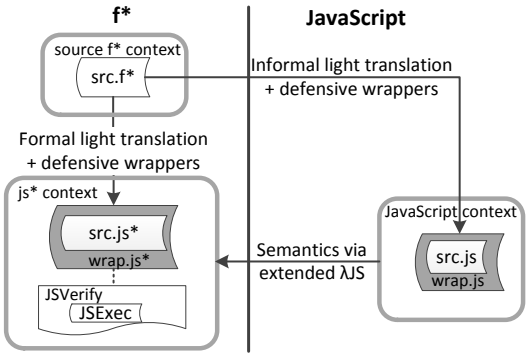


Figure 1. Architecture

This paper presents a correct and secure compiler from a variant of ML with higher-order functions and references to JavaScript. Our main result is *full abstraction*: two programs are equivalent in all source contexts if and only if their translations are equivalent in all JavaScript contexts. Full abstraction is an ideal compiler property, inasmuch as it enables local reasoning on source code, without the need to understand the details of the compiler or the target platform. In our case, programmers can rely on their experience with ML, with static scopes and types, or trust source-level verification tools—and largely ignore the rather tricky semantics of JavaScript.

Which semantics for JavaScript? Compared to ML, the semantics of JavaScript is daunting. There are several different ECMA standards and various implementations (mainly by web browsers) that deviate from the standard in idiosyncratic ways. Maffeis et al. (2008) give an operational semantics for the ECMAScript 3 standard, which while extremely detailed, is also unwieldy in that it is not easily amenable to formal proof or to testing. An alternative approach is to give a semantics via translation to a simpler language, and then to test this translation semantics for compliance with browser implementations. This is the approach of Guha et al. (2010), who give a translation of JavaScript into a mostly standard, dynamically typed lambda calculus called λJS. The translation semantics is convenient for our purposes (e.g., it is executable) and not necessarily less precise or more complex. So, following λJS, we give a semantics to JavaScript by elaboration to F⋆ (Swamy et al. 2011), a variant of ML with richer types. We intend this semantics to capture the main features of the ECMAScript 5 standard, including features like getters and setters that were missing in λJS. Our semantics also includes a number of experimental findings pertaining to implementation-specific features of JavaScript such as the `arguments`, `caller`, and `callee` properties.

A high-level view of the paper Figure 1 outlines our technical development. On the left, we have f⋆, a subset of F⋆ that includes higher-order functions, mutable references, exceptions, and fatal errors, but excludes polymorphism for simplicity. Its semantics is parameterized by a type signature that defines the basic constants available to a program. On the right, we have concrete JavaScript.

Our compiler takes an f⋆ program (src.f⋆) with an arbitrary signature and emits JavaScript syntax in two phases. The first phase (the 'light translation') is compositional and translates f⋆ constructs to the corresponding ones in JavaScript, e.g., function closures to function closures, yielding src.js. For code meant to be executed in untrusted JavaScript contexts, we supplement the light translation with carefully crafted defensive wrappers (wrap.js) to securely import and export values at every source type while preserving the translation invariant.

To reason formally about our compiler, we reflect its output within an arbitrary JavaScript context back into f⋆. Specifically, we employ a variant of the λJS semantics to translate JavaScript to js⋆, an instance of f⋆ with a signature JSExec that provides runtime support for js⋆ programs. Our proof of full abstraction relies on refinement typing to establish several key invariants of the translation. For this typing, we introduce JSVerify, a precise typed model of JSExec expressed using monadic refinement types in f⋆. Then, we develop a new eager-normal-form variant of applicative bisimilarity for contextual equivalence in f⋆ and use it to show the main result of the paper, i.e., that two f⋆ programs are equivalent with respect to an arbitrary f⋆ context if, and only if, their defensively wrapped light translations are equivalent with respect to an arbitrary js⋆ context. We summarize our main contributions below.

- We describe a compiler from f⋆ (§3) to JavaScript, including defensive wrappers to safely mediate interactions between translated programs and their context. (§4)
- We introduce js⋆, a model of JavaScript within f⋆ that includes security-relevant features of ECMAScript 5 and popular JavaScript implementations. (§5)
- We formalize our compiler as a translation from f⋆ to js⋆. We show that it is a forward simulation that preserves a typing and heap invariant. This yields safety and correctness for translations of closed programs executed in isolation. Additionally, by typing, we show that the defensive wrappers support safely exchanging values of various types with an untrusted context. (§6)
- We develop a new co-inductive proof technique for f⋆, with labeled bisimulations to capture the interactions of configurations of related terms with their abstract context, such that bisimilarity coincides with contextual equivalence. (§7)
- We prove our compiler from f⋆ to js⋆ fully abstract. (§8)
- We close with a brief discussion and experimental evaluation of our compiler implementation. (§9)

Disclaimer As usual, full abstraction holds only within our formal semantics of JavaScript, and various side channels may still exist in JavaScript implementations, based, for instance, on stack or heap exhaustion, or timing analysis.

This presentation necessarily omits many details. Additional materials, including a technical report with the full formal development, an F⋆ implementation with a JavaScript back-end, sample source and compiled programs, and an updated F⋆ theory in Coq are available at `http://research.microsoft.com/fstar`.

Related work Programming language abstractions have long been recognized as an essential means for protection (Morris 1973); their secure implementations are often specified as full abstraction (Abadi 1998; Abadi et al. 2002; Abadi and Plotkin 2010; Agten et al. 2012). Conversely, many attacks can be interpreted as failures of abstraction, and several counterexamples to full abstraction exist. For example, Mitchell (1993) notes that Lisp with `FEXPR` has no abstraction contexts, and Kennedy (2006) points out the lack of full abstraction in translations from C# to .NET.

Many powerful co-inductive techniques exist for program equivalence, with various combinations of types, higher-order functions, private mutable state, and exceptions (Sumii and Pierce 2005; Lassen 2005). As discussed in §7, ours combines their features so that bisimulations precisely capture the invariants of wrapped translation within untrusted JavaScript contexts. Although employing logical relations instead of bisimulations, Ahmed and Blume (2008) also use type-directed wrappers to prove that typed closure conversion in the polymorphic λ-calculus is fully abstract. However, unlike us, they do not use the wrappers in the translation itself.

There has been recent work on protecting JavaScript programs from malicious contexts. For example, Taly et al. (2011) apply a dataflow analysis to check that programs in a subset of ECMAScript 5's strict mode (lacking getters and setters) do not leak private data to an adversary. Using this analysis, the authors were able to prove the safety of object-capability idioms used by the Caja (2012) framework that rewrites JavaScript applications to confine security-critical objects, such as the DOM, behind object capabilities. This confinement property is related to the invariant enforced by our wrappers, which we check by typing. Taly et al., however, do not deal with full abstraction.

2. Challenges in secure JavaScript programming

To illustrate the difficulty of writing secure JavaScript code, we naively implement a protection mechanism around a trusted, external function `rawSend` for posting messages to some target domain (Figure 2). By calling `mkSend(rawSend)`, we should obtain a function that enforce the following policy:

- Send messages only to whitelisted URLs (to avoid privacy leaks); of at most 5 characters (to bound resource usage); that includes a secret credential (to identify the service users).
- Do not leak the `rawSend` function or the secret credential, to prevent bypassing our protection mechanism.

Our implementation calls a `sanitize` function, hypothetically provided by some other trusted library. As typical in JavaScript, linking is performed dynamically through the global name-space.

For simplicity, we use our mechanism to protect an anonymous function that prints the message and its target on the console. The resulting protected `send` function is exported to the global name-space and therefore made available to untrusted scripts:

```
send = mkSend(function (target, msg) {
  console.info("Sent " + msg + " to " + target);});
```

In isolation, our code seems to enforce our policy. However, we are going to demonstrate how, by carefully manipulating the context, a malicious script can bypass our protection mechanism. We do not claim any novelty in describing these attacks (Caja 2012; Taly et al. 2011; Meyerovich and Livshits 2010): with these examples, we aim at giving our reader a glimpse at the challenges met by security-conscious JavaScript programmers as well as prepare the ground for our defensive wrappers in §6.

Attack 1: Overwriting global objects Importing objects from the global name-space is risky: by definition, every script has access to this name-space. For instance, a script can maliciously overwrite the `sanitize` function right before calling the `send` operation:

```
sanitize = function (s,msg) { return msg; };
send("http://www.microsoft.com/owa", "too long!");
```

To prevent this attack, we must run `mkSend` before any hostile script (first-starter privilege) and store a private copy of `sanitize` as well as any other trusted library function it may call.

```
function mkSend(rawSend){
  var whiteList = {"http://www.microsoft.com/mail":true,
                   "http://www.microsoft.com/owa":true};
  function newSend(target, msg){
    msg = "[" + secret_credential + "]" + msg;
    if (whiteList[target]){
      rawSend(target,window.sanitize(5,msg));}
    else { console.error("Rejected.");}};
  return newSend ;}
function sanitize(size,msg) {
  return msg.substring(0,size);}
```

Figure 2. Naive implementation of a secure `send`

Attack 2: Dynamic source code inspection Unsurprisingly, hiding secrets in source code is to be avoided. Supposing that we wanted to keep the `whiteList` secret, a malicious script can use the `toString` method on the `mkSend` function to retrieve its content:

```
var targets = mkSend.toString().match(/\'.*?\':true/g)
targets = targets.map(function (s) {
         return s.replace(/\'(.*?)\':true/,"$1") });
```

A mere regular expression matching on the resulting string lets us extract the list of valid targets. This is not, as such, a violation of the specification, yet it is a rather unusual feature.

Attack 3: Redefining `Object` Since JavaScript is a prototype-oriented programming language, one can dynamically modify properties of any object in the system. In particular, one can add a field to the prototype of the canonical object, `Object`, hence extending the white list without even referring to `whiteList` itself:

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

To preclude this attack, we must ensure that any given field is indeed part of the `whiteList` object, and not inherited from its prototype chain. To this end, we could use a safe private copy (obtained by starting first) of the `hasOwnProperty` method.

Attack 4: Side-effectful implicit coercions Part of the complexity of JavaScript comes from its treatment of coercions: should the need arise, objects are automatically coerced at run-time. Instead of a string, one can, for instance, pass an object with a `toString` method that returns a string in the `whiteList` on the first use and another string as the actual send operation is performed:

```
var count = 0;
var target = { toString: function() {
    return count++ == 0 ? "http://www.microsoft.com/owa"
                        : "http://www.evil.com" }};
send(target, "msg");
```

To tame these implicit coercions, we may explicitly check that the input arguments are of the correct type, using the `typeof` operator. Alternatively, we may force a coercion upon receiving the arguments and store the result—this is the case for `msg` in Figure 2.

Attack 5: Walking the call stack Finally, stepping outside ECMA standards, most implementations of the `Function` object provide a `caller` property that points to the current caller of the function being executed. Abusing this mechanism, any callback (such as an implicit coercion or a getter) grants access to the arguments of its caller `newSend`, including `msg` after concatenation with `secret_credential`:

```
var c;
var target = { toString: function toStr () {
  c = toStr.caller.arguments[1].match(/\[(.*?)\]/)[1];
  return "http://www.microsoft.com/mail" }}
send(target, "msg");
```

This code enables one to retrieve the credential by matching the `msg` argument. Similarly, one could retrieve any secret on the call stack. To guard against this attack, we must explicitly clear the `caller` field of our functions before any potential callback.

Our proposal The examples above show that local reasoning about code in JavaScript can be compromised through a variety of attacks. Hence, writing secure code in JavaScript is a hardship: one must take a great deal of attack vectors into account, and one ends up maintaining extremely cumbersome programs, which makes them more error-prone. We propose that programmers instead use a source language with static types and scopes to write security-sensitive code. A compiler should then securely translate the source language to JavaScript, freeing the programmer from thinking about the subtle semantics of JavaScript. In this context, ML appears to be a particularly effective source language: it has static types and scopes; it is functional, so we can rely on closures and higher-order functions also available in JavaScript; and, being impure, we can adopt a programming style that approaches idiomatic JavaScript. In ML, the example of Figure 2 can be written as shown below, which clearly meets the stated security goals.

```
let mkSend rawSend =
  let whiteList = ["http://www.microsoft.com/mail";
                   "http://www.microsoft.com/owa"] in
  fun target msg →
    let msg = "[" ^ secret_credential ^ "]" ^ msg in
    if mem target whiteList then rawSend target (sanitize 5 msg)
    else consoleError "Rejected."
```

3. Syntax and semantics of f⋆

In this paper, we use f⋆, a fragment of F⋆ (Swamy et al. 2011) similar to ML, with the syntax shown below and a standard, small-step, call-by value semantics (see the full paper). We have extended the original presentation and formal development of F⋆ with exceptions, fatal errors, and primitive support for a mutable store.

Values range over variables, memory locations, abstraction over terms, and n-ary, fully applied data constructors. We add a form of results r, which, in addition to values, includes exceptions **raise** v and fatal **error**. Expressions are in a partial administrative normal form, with, for instance, function application $e\ v$ requiring the argument to be a value. We also have pattern matching, reference allocation, assignment and dereference, and exception handlers.

Syntax of f⋆

v	::=	$x \mid \ell \mid \lambda x{:}t.e \mid D\ \bar{t}\ \bar{v}$	values
r	::=	$v \mid \textbf{raise}\ v \mid \textbf{error}$	results
e	::=	$r \mid e\ v \mid \textbf{let}\ x = e\ \textbf{in}\ e' \mid v_1 := v_2 \mid \mathsf{ref}\ v \mid !v \mid$ $\textbf{try}\ e\ \textbf{with}\ x.e \mid \textbf{match}\ v\ \textbf{with}\ D\ \bar{\alpha}\ \bar{x} \to e\ \textbf{else}\ e'$	terms
t	::=	$T \mid \mathsf{ref}\ t \mid t \to t'$	types
H	::=	$\cdot \mid (\ell \mapsto_t v) \mid H, H'$	store
$F[\_]$	::=	$\_ \mid F\ v \mid F\ t \mid \textbf{let}\ x = F\ \textbf{in}\ e$	exn. ctx
$E[\_]$	::=	$\_ \mid E\ v \mid E\ t \mid \textbf{let}\ x = E\ \textbf{in}\ e \mid \textbf{try}\ E\ \textbf{with}\ x.e$	eval. ctx
S	::=	$\cdot \mid D{:}t \mid T{::}\kappa \mid S, S'$	signature
Γ	::=	$\cdot \mid \Gamma, x{:}t \mid \Gamma, \alpha \mid \Gamma, \ell{:}t \mid \ldots$	type env.

An f⋆ runtime state, written $H \mid e$, is a pair of a store mapping locations to values and a program expression. The reduction relation has the form $H \mid e \to_S H' \mid e'$ where the index S is a fixed inductive signature that defines all constructors. This signature includes at least a type `exn` for exceptions, types `ref` t for references, and `unit`. We also freely use common primitive types like `int` and `bool`, and expect these to be in the signature as well. Our syntax does not include a fixpoint form because recursion can be encoded with recursive datatypes in the signature. We consider several instantiations of the signature S in this paper, to define our source language (§4) and to embed dynamically typed JavaScript within f⋆ (§5).

Syntactic sugar We write applications $e\ e'$ as abbreviations of **let** $x = e'$ **in** $e\ x$, for some fresh x. A similar transformation applies to pattern matching, reference operations, exception raising, etc. We write **if** e **then** e_1 **else** e_2 for **match** e **with** `true` $\to e_1$ **else** e_2, and $e_1; e_2$ for **let** $\_ = e_1$ **in** e_2. Additionally, in code listings, we rely on the concrete syntax of F⋆, which closely resembles OCaml and F#.

Plain types F* includes various dependent typing features, but we ignore this in f*, and restrict the types to a monomorphic subset of ML including function types $t \to t'$, references, and recursive datatypes. Nevertheless, we have extended our Coq-based metatheory of the full F* language to include exceptions, state and errors, and proved subject reduction for the reduction of open terms, i.e., terms that may contain free variables, which is used in §7 and §8. We present a specialized version of this theorem below, where we use the type judgment for F* runtime states. This is written here as $S;\Gamma \vdash H \,|\, e : t$, denoting that in an environment including the signature S, free variables Γ, and the typed domain of H (written $\sigma(H)$, including $\ell : t$ for each $\ell \mapsto_t v$ in H), the store H is well-typed and the expression e has type t. When the signature S is evident from the context, we simply write $\Gamma \vdash H \,|\, e : t$.

Theorem 1 (Type soundness for open-term reduction)**.** *Given S, Γ, H, e, and t such that $S;\Gamma \vdash H \,|\, e : t$, either (1) e is a result; or (2) e is an open redex, i.e., $e \in \{E[x\ v], E[\textbf{\textit{match}}\ x\ \textbf{\textit{with}} \ldots], E[x := v], E[!x]\}$; or (3) there exist H', e' such that $H \,|\, e \to_S H' \,|\, e'$ and $S;\Gamma \vdash H' \,|\, e' : t$.*

Contextual equivalence We only observe well-typed terms and compare them at the same plain types. Following Theorem 1, we define basic observations on runtime states s: (1) s returns, that is, $s \to_S^* H \,|\, r$, with three kinds of results: any value (written $s \Downarrow$); any exception (written $s \Downarrow$ **raise**); or an error (written $s \Downarrow$ **error**); or (2) s diverges (written $s \Uparrow$) when it has an infinite sequence of reductions; or (3) (only in case s is open), s reduces to a redex with a free variable in evaluation context.

We define contextual equivalence for closed values and expressions, considering all typed evaluation contexts. (Equivalence between open terms can be re-stated using closed function values.)

Definition 1 (Contextual Equivalence)**.** *Two (closed, typed) runtime states s and s' have the same behavior, written $s \approx_{\mathsf{e}}^{\bullet} s'$, when they either both return the same kind of result, or both diverge.*

Two (closed, typed) terms are equivalent, written $e \approx_{\mathsf{e}} e'$, when they have the same behavior in all evaluation contexts.

4. A compiler from f* to JavaScript

We present our compiler from f* to JavaScript, using the f* program below as a running example.

```
let mkLog _ = let log = ref (Nil : list string) in
              let add x = log := Cons x !log in
              let iter f = List.iter f !log in
              (add, iter)
```

Calls to mkLog return an abstract interface to a log, with functions add and iter to extend the log with a string and to operate on its contents, respectively. Reasoning in f*, it is clear for instance that the log only contains strings and that it grows monotonically.

In this section, we illustrate informally how our compiler ensures that all source invariants are preserved in the translation to JavaScript. In subsequent sections, we justify it by formalizing our compiler as a translation from f* to js* and proving it fully abstract.

The light translation Our compiler proceeds in two phases. The first phase is compositional and purely syntax-directed: the translation function $[\![e]\!]$, shown in Figure 3, translates f* constructs to their JavaScript counterparts. Following standard practice, we assume uniqueness of variable names. We also use an auxiliary function, $locals(e)$, that collects the let- and pattern-bound names not enclosed within additional λs of expression e.

We translate functions in f* to functions in JavaScript as follows: Local variable declarations in JavaScript always occur at the top of a function body, so we collect the source locals and declare them upfront. When reading a variable x (the second rule), we simply lookup the JavaScript variable with the same name `x`. In the third rule, we translate let-bindings to JavaScript *sequence expressions* `(x=e0,e1)`—in JavaScript, this expression evaluates `e0`, assigns the result to `x`, and then evaluates `e1`.

$[\![\lambda x{:}t.e]\!] \mapsto$ `function(x){`$\overline{\texttt{var y;}}$`return`$[\![e]\!]$`;}` where $\bar{y} = locals(e)$

$[\![x]\!] \mapsto$ `x`

$[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] \mapsto$ `(x=`$[\![e_1]\!]$`,`$[\![e_2]\!]$`)`

$[\![e_1\ v_2]\!] \mapsto [\![e_1]\!]([\![v_2]\!])$

$[\![D\ \bar{v}]\!] \mapsto$ `{"tag":`$str\ D$`,`$\overline{str\ i\texttt{:}[\![v_i]\!]}$`}`

$[\![\textbf{ref}\ v]\!] \mapsto$ `{"ref":`$[\![v]\!]$`}`

$[\![v_1 := v_2]\!] \mapsto ([\![v_1]\!]$`.ref=`$[\![v_2]\!]$`,undefined)`

$[\![!v]\!] \mapsto [\![v]\!]$`.ref`

$[\![\textbf{error}]\!] \mapsto$ `alert("error")`

$[\![\textbf{match}\ v\ \textbf{with}\ D\ \bar{x} \to e_1\ \textbf{else}\ e_2]\!] \mapsto$
`(`$[\![v]\!]$`.tag ===`$\ str\ D$`)?(`$\overline{\texttt{x}_i\texttt{=}[\![v]\!]\,[str\ i]}$`,`$[\![e_1]\!]$`):`$[\![e_2]\!]$

Figure 3. Light translation from f* to JavaScript

Function application is straightforward. We translate data values $D\ v_0 \ldots v_{n-1}$ to objects with a `tag` field recording the name of the constructor as a string, and with fields `"0"`, ..., $str\ (n-1)$ containing the translations of the arguments—the meta-function str encodes its argument as a JavaScript string. References have a single field `"ref"`. When assigning a reference, we update the `ref` field and then evaluate to `undefined`, our representation of the `():unit` value in JavaScript. We model fatal error in JavaScript by calling the `alert` function, which pops up a dialog in most browser environments—several other possibilities exist for modeling fatal errors. Finally, we translate matching to JavaScript *conditional expressions*, `e?e0:e1`. Observe that the only statement forms we use are within functions, where we use `var` declarations and `return`. By relying only on the expression forms of JavaScript, we obtain a simple compositional translation.

For simplicity, the input of the translation does not contain exceptions and their handlers, but we still study their properties for all f* evaluation contexts (including exceptions). Technically, we also require that f* does not have `ref unit` and similar types whose values are all contextually equivalent in f* but whose translations may be distinguished in JavaScript using untyped equality. Finally, we do not formalize the translation of polymorphic data constructors, although they are supported by our compiler implementation.

At top level, our formalization applies to the translation of programs enclosed within a function. Our implementation augments this with simple handling for top-level let-bindings. Running this on the declaration of mkLog, we obtain the following JavaScript, where `List.iter` refers to the translation of the f* function List.iter.

```
function mkLog (u) {
 var log; var add; var iter;
 return
   (log={"ref":{"tag":"Nil"}},
   (add=(function(x){
     return (log.ref={"tag":"Cons", "0":x, "1":log.ref},
             undefined); }),
   (iter=(function(f){
     return (List.iter(f)(log.ref));}),
   {"tag":"Pair", "0":add, "1":iter})));}
```

Type-directed defensive wrappers Providing a JavaScript context with direct access to `mkLog` is not fully abstract: an adversary could, for example, call `mkLog`, obtain `iter`, and then call it with a function `f` that (as in attack 5 of §2) walks the stack, accesses `log` directly from the arguments of `List.iter`, and breaks its invariants. To protect our code, we apply type-directed wrappers that build a firewall between the lightly translated f* code and its context.

Our wrappers are JavaScript functions indexed by source types t. They come in pairs: a 'down' wrapper, written $\downarrow t$, takes a light translation of a source value $v{:}t$ and exports it safely to the context; an 'up' wrapper, written $\uparrow t$, takes any JavaScript value supplied by the context and attempts to extract from it a value that is a light translation of some source $v{:}t$; this may fail.

In addition to ensuring that the translated f* code and its context interact at the expected types, the wrappers seek to enforce a strict heap separation between the code and the context. Specifically, we

```
function downunit(x) { return x;}
function upunit(x) { return undefined;}
function downbool(x) { return x;}
function upbool(z) { return (z ? true : false);}
function downstring(x) { return x;}
function upstring(x) { return (x + "");}

function downpair(dn_a, dn_b) {
  return function (p) {
    return {"tag":"Pair",
            "0":dn_a(p["0"]),"1":dn_b(p["1"])};}}
function uppair(up_a, up_b) {
  return function(z) {
    return {"tag":"Pair",
            "0":up_a(z["0"]),"1":up_b(z["1"])};}}

function downfun (up_a,down_b) {
  return function (f) {
    return function (z) {
      return (down_b (f (up_a(z)))); }}}
function upfun (down_a,up_b) {
  return function (f) {
    return function (x) {
      var z = down_a(x);
      var y = undefined;
      function stub(b) {
        if (b) { stub(false); }
        else { y = up_b(f(z)); } }
      stub(true); return y; };};}
```

Figure 4. Selected wrappers in JavaScript

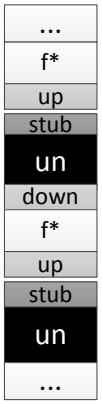
ensure that the context never obtains a direct reference to an object that is used by the light translation; references from f* objects to objects owned by the context (we call such objects untrusted, or un, objects) are also problematic, since the contents of un objects are unreliable, e.g., they may change unexpectedly. So, access to f* objects by the attacker, and vice versa, are mediated by wrappers.

Figure 4 lists some wrappers used by our compiler. For immutable base types shared between f* and JavaScript, such as strings, the 'down' wrapper does nothing, whereas the 'up' wrapper forces a coercion. There are various JavaScript idioms that serve to induce coercions at particular types, e.g., for Booleans, we use an explicit conditional expression; for numbers, we use unary addition; for strings, we concatenate with the empty string, etc. This ensures, for instance, that `true` and `false` are indeed the only imported Boolean values, foiling problems like attack 4 from §2.

For datatypes such as pairs and lists (and any allocated data), we must ensure that wrapping preserves heap separation. Thus, we allocate a fresh representation and recursively wrap their contents. The 'up' wrapper is safe even as its code accesses fields (which may trigger callbacks to the context via implicit coercions or getters) because the imported content is kept as a local value on the 'up' wrapper stack. Our code includes wrapper generators; for instance, `downpair` takes as parameter two 'down' wrappers for types a and b and returns a 'down' wrapper for pairs containing an a and a b.

For functions, the situation is more complex, since the 'up' wrapper has no way to check that its argument is the valid representation of a source f* function. Instead, the wrapping is deferred: the function `downfun`, corresponding to $\downarrow(a \to b)$, exports a function `f` by wrapping it with another function that first imports the argument `x`, then applies `f`, and finally exports the result. In the other direction, one might have expected `upfun` (for $\uparrow(a \to b)$) to be strictly dual to $\downarrow(a \to b)$, i.e., export the argument, apply the function, and import the result. However, this is insufficient. As attack 5 of §2 illustrates, the JavaScript calling convention provides a function with access to the function object and arguments of its caller. If a trusted function were to call an untrusted one directly, the latter obtains a reference to the arguments of the former, breaking our heap separation discipline.

To this end, following the code of `upfun` in Figure 4, the wrapper for importing an untrusted function `f` (purportedly the translation of an $a \to b$ value) is itself a function, callable from any trusted context, that first exports its argument into a local variable `z`, then calls a fresh, single-use `stub`. The `stub` makes the call to `f` on behalf of the trusted code, but before doing so, it needs to prevent the context from using its own `caller` field to traverse the call stack. Unfortunately, directly clearing this field (e.g., by setting `arguments.callee.caller = undefined`) is not supported by many browsers—assignments to the `caller` property of a `Function` are silently ignored. Instead, we make `stub` call itself once before calling `f`. Since `Function` objects in JavaScript are shared across all the activations of a function, this recursive call induces a cycle in the chain of `caller` properties, ensuring that when the call to `f` proceeds, the context obtains a reference to the `stub` but cannot walk the stack beyond the stub and compromise trusted code. After the untrusted call completes, `up_b` wraps up the result and stores it in `y`. (Returning the value directly is dangerous, since the attacker has a pointer to the `stub` closure, so, it may be able to call this closure later and receive the protected value.) After the `stub` completes, the wrapper returns the contents of the local variable `y`. Thus, to an attacker that attempts to traverse the call stack via the `caller/callee` properties, the stack (growing downward) appears as depicted alongside. Walking upward, untraversable `stub` objects delimit regions of the stack that transition from untrusted to trusted code. Additionally, the up and down wrappers mediate all calls across trust boundaries.



Top-level translation Continuing with our example, we list below the script that makes `mkLog` available to an arbitrary JavaScript context after suitable wrapping. Rather than placing `mkLog` directly into the global name-space (i.e., the `window` object in a web browser), our compiler generates a function `init` that takes the `window` object as a parameter, defines the lightly translated code of mkLog in a local variable, and exports it to `window.mkLog` after unfolding and applying the down-wrapper $\downarrow(\mathsf{unit} \to (\mathsf{string} \to \mathsf{unit} * (\mathsf{string} \to \mathsf{unit}) \to \mathsf{unit}))$. After running `init(window)`, our script overwrites `init` to prevent any later use (such as `init.toString()`).

```
function init(w) {
  function mkLog(u){...} // light translation shown above
  w.mkLog=
   downfun(upunit,
           downpair(downfun(upstring, downunit),
                    downfun(upfun(downstring, upunit),
                            downunit)))(mkLog); }
init(window); init=undefined;
```

Threats and countermeasures We briefly review potential threats to full abstraction, and informally discuss how we handle them.

- Modifying `Object.prototype` can override the default behavior of objects, e.g. when accessing their properties. As an invariant, translated f* code never triggers a prototype chain traversal, so our translation does not depend on `Object.prototype`.
- By changing `Function.prototype.apply`, an adversary can interpose code at function calls. However, ECMAScript 5 states that this interception does not affect primitive function calls.
- `Function.toString` returns the static source of a function closure as a string. Our wrappers ensure that, for any function `g` handed to the adversary, `g.toString()` always returns the text of its down wrapper, that is, the constant string `"function (z) { return (down_b (f (up_a(z)))); }"`

- Implicit coercions are pre-empted by systematically forcing coercions in 'up' wrappers.
- Stack walks via `callee` and `caller` properties are countered by the stub mechanism described above.
- Some browsers provide `(new Error()).stack`, which dumps the current stack as a string. Assuming that our code runs first, we disable it in the `init` function, using the code below.

```
var oldErr=Error; window.Error=function(){
  var x=oldErr(); x.stack=""; return x;};
```

Are these countermeasures sufficient? The rest of the paper, culminating with the main results of §8, provide a positive answer, at least within our semantics of JavaScript.

5. A semantics of JavaScript in F*

We begin our formal development with a semantics of JavaScript by translation to js*, the instance of F* with inductive signature JSExec described below; this allows us to carry out our full-abstraction argument entirely within a single language. We base our semantics on λJS, a dynamically typed language to which Guha et al. (2010) translate JavaScript. We extend λJS to include some features of ECMAScript 5 that were missing in the original formulation (which targeted ECMAScript 3), as well as browser-specific features that are relevant for full abstraction. Concurrently, Politz et al. (2012) have extended λJS to cover the strict mode of ECMAScript 5.

We focus on a few main features of js*: dynamic typing, object properties, function creation, the calling convention, control operators, and `eval`. We refer to our technical report for a complete presentation, including a formal translation from λJS to js*.

Dynamic typing. In order to type any JavaScript values, JSExec defines dyn, a standard 'type dynamic', as follows.

```
type dyn = Null : dyn | Undef : dyn | Bool: bool →dyn
  | Str: string →dyn | Num: float →dyn
  | Obj: loc →dyn | Fun: dyn →(dyn →dyn →dyn) →dyn
and obj = list (string * property)
and loc = ref obj
and property = Data: attrs →dyn →property
  | Accessor: attrs →(dyn * dyn) →property
type exn = Break: int→dyn→exn | Exn: dyn→exn | Return: dyn→exn
```

The type dyn has a constructor for each JavaScript primitive type. For instance, the JavaScript string literal `"Rome"` is represented as Str `"Rome"`: dyn. Objects are references to maps from string (property names) to property, the type of values or accessors (getters and setters). Their property attributes attrs specify, for instance, whether they are writable or enumerable. (Our translation does not rely on attributes for security.) Getters and setters are treated as functions, called to perform property lookups or assignments. Functions in JavaScript are also objects—one may set properties on them, writing, for instance, `function foo(){}; foo.x = 17`. To handle this, we represent JavaScript functions as js* values Fun o f, constructed from a function object o:dyn and a closure f. All functions in JavaScript receive an implicit `this` parameter and, following λJS, a single argument object with a field for each of their explicit arguments. Thus, the closure f within Fun o f has type dyn →dyn →dyn. We discuss the three kinds of exceptions shortly.

Function creation and application While outside the ECMAScript specification, most browsers implement a quirk in their calling convention. Functions `f` receive their (variable number of) arguments in single `arguments` objects. These objects include a `callee` field that points to the function object of `f`. Conversely, function objects include an arguments field that points back to the argument object of their last activation, if any, and a caller field that points back to the function object of their last caller. (This field may point to `f` itself, if it makes recursive calls, or be `null`, for top-level calls.) These fields are implicitly updated at every call. In particular, all JavaScript functions are recursive through the store, since they are given access to their own object.

To model this calling convention, JSExec defines several operations. First, a lookup function looks up the property name f in the map of an object (accounting for function values as well).

```
let lookup (d:dyn) (f:string) = match d with
 Obj loc | Fun (Obj loc) _ →assoc f !loc | _ →None
```

Similarly, a modify function updates properties within object maps. In this section, we use shorthands for these functions:

$e\langle f\rangle \triangleq$ **match** lookup e f **with** Some(Data _ x)→x |_→Undef

$e_1\langle f\rangle = e_2 \triangleq$ modify e_1 f e_2

To allocate functions, JSExec defines mkFun as follows:

```
let mkFun (s:string) (code: dyn →dyn →dyn →dyn) =
  let o = alloc () in let f = Fun o (code o) in
  o⟨"@code"⟩ = f; o⟨"@toString"⟩ = Str s; o⟨"prototype"⟩ = ...; ...; f
```

For instance, we formally translate `function (x){ return x;}` to

```
mkFun "function(x)..." (fun o this args →select o args "0")
```

where the first argument is a string literal that represents the source text of the function and the second argument is a js* closure that receives three objects: the (soon-to-be-created) function object o, the 'this' parameter, and the actual arguments. The call to mkFun allocates o, partially applies the closure to o, and sets various properties on o before returning f.

To call functions, JSExec provides apply, which receives four arguments: caller, the object of the calling function; callee, the function to be called; a this pointer; and an args object.

```
let apply (caller:dyn) (callee:dyn) (this:dyn) (args:dyn) : dyn =
  match callee with | (Fun o f) →
    let caller0 = o⟨"caller"⟩ in let args0 = o⟨"arguments"⟩ in
    try o⟨"caller"⟩ = caller⟨"@code"⟩; o⟨"arguments"⟩ = args;
      args⟨"callee"⟩ = callee; f this args
    with Break _ →error | e →raise e
    finally (o⟨"caller"⟩ = caller0; o⟨"arguments"⟩ = args0) | ...
```

Following the code, apply calls f with argument this and args. First, however, it saves the callee's caller and arguments fields, sets these fields for the current call, and sets a pointer from args to callee. Conversely, once the call returns, apply restores the callee's fields to their old value. (The derived F* form **try** ... **with** ... **finally**, detailed below, ensures that the fields are restored even if the call raises an exception.) Experimentally, this reflects major browser implementations of JavaScript.

Property access In JavaScript, properties of objects are looked up first in the object's own property map, then in the object's prototype (stored in a special property "@proto"), walking the prototype chain as needed. Once found, if the property happens to be a getter, then the getter function is called, otherwise its value is returned. This is implemented by the select function, shown below. Since calling the getter requires passing a caller object, we write select caller l f to select field f from object l in the context of the function object caller. (Recall that the translation of our example function included a call to select passing its object o as a parameter.)

```
let rec getProperty (l:dyn) (f:string) = match lookup l f with
  | Some p →Some p
  | None →match lookup l "@proto" with
      | Some (Data _ l') →getProperty l' f | None →None
let select (caller:dyn) (l:dyn) (f:string): dyn =
  match getProperty l f with
      | Some(Accessor _ (g, _)) →apply caller g l (mkEmptyArgs())
      | Some(Data _ d) →d | _ →Undef
```

A similar function, update caller l f v, sets property f on object l to value v. This function traverses the prototype chain looking for a setter for property f. If a setter is found, update calls it with caller and v; otherwise it calls modify l f v.

Exceptions and control operators We model exceptions and the other imperative control operators of JavaScript using f⋆ exceptions. JavaScript has a `return` statement to end the current call, and a `break l` statement to return control to the code location labeled `l`, which must be defined within the same function body. λJS desugars both to a single form, which we represent as the exception Break l v. Additionally, we use exceptions Exn v for JavaScript exceptions, and Return v to encode finally clauses, as follows.

try e_1 **with** e_2 **finally** $e_3 \triangleq$ **try** (**try** e_1 **with** e_2) **finally** e_3
try e_1 **finally** $e_2 \triangleq$ **try raise** (Return e_1)
with y. **match** y **with** | Return r $\to e_2$; r | _ $\to e_2$; **raise** y

Dynamic evaluation JSExec does not support `eval`, as this would involve parsing, translating, and loading js⋆ code at run time. On the other hand, js⋆ contexts can implement any 'primitive' function eval using the datatypes of JSExec, together with any values they obtain by interacting with our wrapped translation. As such, our full-abstraction result applies also to contexts that use `eval`.

Typability We conclude this section with a simple result: every λJS program translated to js⋆ is well-typed against JSExec. In the statement below, ⟦e⟧ is the translation of a λJS expression to js⋆.

Theorem 2 (Typability of λJS translation)**.** *For all λJS programs* e *with free variables* $\overline{x}$*, we have JSExec*; $\overline{x{:}\mathsf{dyn}} \vdash [\![e]\!] : \mathsf{dyn}$.

Contextual equivalence in JavaScript (discussion) After translation to js⋆, we formally compare JavaScript programs using the contextual equivalence of §3 (Definition 1). This equivalence relies on the observation of fatal errors, which are not primitive in JavaScript, but informally account for any code with an immediate observable effect, such as `alert("error")` or `window.location = "http://error.com"`. This equivalence in js⋆ is also *a priori* finer than JavaScript equivalence, inasmuch as it quantifies over all well-typed js⋆ contexts, not just those obtained by translating JavaScript contexts. Thus, we err on the safe side: most of our results would apply unchanged for variants and extensions of JSExec (as long as its signature is unchanged), for instance, to model additional features of JavaScript implementations. Conversely, §8 shows that translations of JavaScript contexts are complete at least for interacting with wrapped translated f⋆ programs.

6. A type-preserving simulation from f⋆ to js⋆

Formally, our compiler can be viewed as the translation from f⋆ to JavaScript (§4) composed with the embedding of JavaScript into js⋆ (§5). In this light, its correctness is far from obvious. For example, even though superficially we translate f⋆ functions to JavaScript functions, several corner cases of their semantics lurk beneath the surface syntax. As we have seen, functional values translate to expressions that allocate several objects, and are subject to a calling convention with side-effects.

This section proves several safety properties for the f⋆-to-js⋆ compiler. In order to carry out these proofs, we use an alternative, monadic type system for f⋆ due to Schlesinger and Swamy (2012), as well as an application of this type system to JavaScript provided by Swamy et al. (2012). Specifically, we use JSVerify, a variant of JSExec with monadically refined types that allows us to state and prove precise typing and heap invariants of js⋆ programs. Using this machinery, we prove that the light translation preserves types and is a (weak) forward simulation. Additionally, we prove that the defensive wrappers successfully maintain several key invariants, including separating un objects from the others. While useful in their own right for whole programs (e.g., we can prove that when a source f⋆ program has no assertion failures, then neither does its translation), these properties serve primarily as lemmas that facilitate the main results of §8.

Syntax of types in monadic f⋆

t	::=	$T \mid a \mid t\ t \mid x{:}t\{\phi\} \mid \forall a{::}\kappa.t \mid x{:}t \to t \mid \overline{x{:}t} \to \mathsf{DST}\ t\ \phi$	types
ϕ	::=	$\top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Longrightarrow \phi' \mid \forall x{:}t.\phi \mid \exists x{:}t.\phi$	formulas
	|	$P \mid \lambda x{:}t.\phi \mid \Lambda a{::}\kappa.\phi \mid \phi\ u \mid \phi\ \phi$	
κ	::=	$\star \mid E \mid x{:}t \Rightarrow \kappa \mid \alpha{::}\kappa \Rightarrow \kappa$	kinds
u	::=	$v \mid t \mid \mathsf{V}\ v \mid \mathsf{E}\ v \mid \mathsf{err} \mid \mathsf{Sel}\ u\ u \mid \mathsf{Upd}\ u\ u\ u \mid u_1 + u_2 \ldots$	logic term

Monadic f⋆ The type system of monadic f⋆ is based on a monad of predicate transformers called the *Dijkstra state monad*. The type of this monad is written DST t wp, and stands for stateful computations yielding a t result and with specification described by the weakest pre-condition predicate transformer wp. The transformer wp takes a post-condition formula post, relating a result of type t and a final heap, and returns a pre-condition formula pre, a predicate on an initial heap. Given a program e : DST t wp and a particular post-condition post to be proven, the f⋆ type checker builds a verification condition wp post and uses Z3, an SMT solver (de Moura and Bjørner 2008), to try to discharge the proof.

We give the syntax of monadic f⋆ types above. As in §3, the type language is parameterized by a signature S that also defines a set of type constructors T. Types include variables a, type applications $t\ t'$, refinement types $x{:}t\{\phi\}$, and polymorphic types $\forall a{::}\kappa.t$. Data constructors are n-ary and are given pure dependent function types $\forall\overline{a{::}\kappa}.\overline{x{:}t} \to t'$. General function types have the form $\overline{x{:}t} \to \mathsf{DST}\ t'\ \phi$, with a monadic co-domain dependent on the curried arguments $\overline{x{:}t}$.

Formulas ϕ include the usual connectives (implication is written $\Longrightarrow$, distinguishing it from the kind constructor $\Rightarrow$ discussed below). Predicates P may be interpreted (e.g. equality), although uninterpreted predicates can be introduced by the signature S. Formulas also include a strongly normalizing applicative language of functions over logical terms and other predicates. We write $\lambda x{:}t.\phi$ and $\Lambda a{::}\kappa.\phi$ for predicate literals or, in the latter case, for transformers from a predicate (or type) a of kind κ to ϕ. Formulas can be applied to other formulas or to logical terms u.

The type system is parametric in the logic used to interpret formulas. By default, we use a first-order logic with uninterpreted functions and theories of functional arrays, arithmetic, datatypes, and equality. For example, we use type heap and interpreted functions Sel: heap → ref α → α and Upd: heap → ref α → α → heap from the select/update theory of functional arrays (McCarthy 1962) to model the store. Logic terms also include three kinds of result constructors: V v is a result value; E v is an exceptional result; and err is the error value. We define *asResult* $v \triangleq \mathsf{V}\ v$, *asResult* (**raise** v) $\triangleq \mathsf{E}\ v$, and *asResult* **error** $\triangleq \mathsf{E}$. Additionally, we write Result r ϕ as an abbreviation asserting that ϕ x is valid when r = V x.

We have two base kinds: $\star$ is the kind given to value types, while E is the kind of types that stand for erasable specifications, with the sub-kinding relation $\star \leq E$. We include dependent function kinds, both from types to kinds, and from kinds to kinds. In most cases, unless we feel it adds clarity, we omit writing kinds.

The main judgments in the monadic type system are $S;\Gamma \vdash^D v : t$ for values and $S;\Gamma \vdash^D e : \mathsf{DST}\ t\ \phi$ for expressions. They rely on an auxiliary judgment, $S;\Gamma \models \phi$, stating that formula ϕ is derivable from the logical refinements of context Γ. For example, we can type the program $\lambda x{:}$ref int.$x :=\!! x + 1$ as x:ref int → DST unit Λpost.λh:heap.post (V ()) (Upd h x ((Sel h x) + 1)). This is the type of a function from integer references x to unit, where the weakest pre-condition for any predicate post relating a unit result to the output heap is the formula post (V ()) (Upd h ((Sel h x) + 1)), indicating that the function always returns normally with () and updates the input heap h with the contents of location x incremented.

The judgments for typing terms extend naturally to a judgment on runtime states, written $S;\Gamma \vdash^D H \mid e : \mathsf{DST}\ t\ \phi$. The soundness theorem for monadic f⋆ is a refinement of Theorem 1; it also applies

to open reductions. In its statement below, we rely on a function *asHeap* that interprets a store H as a Sel/Upd value of type heap.

Theorem 3 (Monadic soundness)**.** *Given S, Γ, H, e, t, ϕ such that $S;\Gamma \vdash^D H \mid e : DST\ t\ \phi$, and a post-condition ψ such that $S;\Gamma \models \phi\ \psi\ (asHeap\ H)$ is derivable; either: (1) e is a result and $S;\Gamma \models \psi\ (asResult\ e)\ (asHeap\ H)$; (2) e is an open redex; or (3) there exist H', e', ϕ', such that $H \mid e \rightarrow_S H' \mid e'$, $S;\Gamma \vdash^D H' \mid e' : DST\ t\ \phi'$ and $S;\Gamma \models \phi'\ \psi\ (asHeap\ H')$.*

We sometimes use a monad iDST, an abbreviation for the DST monad augmented with a heap invariant: iDST t wp is the type of a computation that, when run with an input heap h satisfying HeapInv h ∧ wp post h, diverges or produces an output heap h' and result r:t satisfying HeapInv h' ∧ DeltaHeap h h' ∧ post r h'. We describe the heap invariants enforced by iDST below, referring to our online material for the full definitions.

JSVerify: a precise typed model of JSExec We now present a few key elements in JSVerify, an interface for JSExec written using the precise types of monadic f⋆. We start by showing how we recover the precision of the source type system by refining the type dyn introduced in §5. A central feature of this refinement is JSVerify's model of a partitioned js⋆ heap. We conclude this subsection with a lemma relating JSExec to JSVerify.

Dynamic typing We structure our formal development by translating the types of f⋆ into logical formulas. Specifically, we use a refinement of type dynamic developed by Swamy et al. (2012) to embed the simple type language of f⋆ within the refinement logic of monadic f⋆. For example, rather than typing Str "Rome" simply as dyn, we type it as x:dyn{TypeOf x = string}, where TypeOf is an uninterpreted function from values to E-kinded types.

We show a few cases in the definition of type dyn used in JSVerify below. The full listing of JSVerify is available online.

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

As in JSExec, an object is a value Obj l, for some heap reference l:loc. In addition, the refinement on the Obj constructor recalls the refinement on the underlying loc. The Fun constructor takes two value arguments, an object o and a function closure, as before. In addition, it now takes a specificational argument: a predicate transformer wp for the function closure. The refinement on the argument o requires it to be an object (in addition to some other properties, which we elide from this presentation). The function closure is itself typed within the iDST monad with weakest precondition wp. The type of Fun recalls the predicate transformer of the closure in its result type, i.e., TypeOf d=WP wp where WP is a type-level coercion from the kind of wp to E.

Invariants of a partitioned heap Our proof strategy involves enriching the type of heap references to keep track of a logical partition of the js⋆ heap into five compartments. This partition allows us to prove several invariants, e.g., that there are no references from objects in the attacker-controlled part of the heap to elsewhere. These five heap compartments are as follows:

Inv*: the invariant heap* Let-bound variables, arguments and data are immutable in f⋆ but are held in heap locations in js⋆. To keep track of these locations, we place them in a logical compartment called the Inv heap. A complication that we handle is that these locations are not strictly immutable—JavaScript forces us to preallocate locals, requiring a mutation after allocation, and the calling convention also involves implicit effects. Still, we prove that, once set, all the relevant fields of objects in the Inv heap never change.

Ref*: the heap of source references* Locations used to represent the translation of f⋆ values of type ref t are placed in the Ref heap, where an invariant ensures that the content of a Ref heap cell, once initialized, always holds a translation of a t-typed source value.

Abs*: the abstract heap of function objects* Recall that every function in js⋆ is associated with a heap-allocated object whose contents is updated at every function call. We place these unstable locations in the Abs heap, and ensure that translated source programs never read or write from these locations, i.e., function objects are abstract.

Un*: the untrusted heap* This heap compartment is used to model locations under control of the attacker. Our full-abstraction result relies crucially on a strict heap separation to ensure that locations from the other compartments never leak into the Un heap (with one exception, discussed next).

Stub*: the heap of declassified function objects* Function objects corresponding to stubs in the upfun wrapper are allocated in a compartment of their own. These objects initially belong to the f⋆ translation, but, once used, they become accessible from the Un heap. To this end, we implement (and type) a logical declassification protocol, allowing us to prove that, as their ownership is transferred to the attacker, stub objects do not leak any information.

To keep track of these heap compartments, JSVerify enriches the representation of heap allocated objects with (ghost) metadata, outlined below. We have a tag for each compartment. The type tobj, which is a wrapper around the type obj, associates it with its tag and a predicate p, an invariant on the contents of the object.

```
type tag = Inv: tag | Ref: tag | Abs: tag | Un: tag | Stub: tag
type tobj = TO: ∀p::obj ⇒ E. t:tag
                → o:obj{(t=Ref ⟹ "ref" ∈ dom o ∧ p o)
                        ∧ t =Inv ⟹ ...}
                → v:tobj{TypeOf v=PT p t}
type loc = TL: ∀p::obj ⇒ E. t:tag → ref (v:tobj{TypeOf v=PT p t})
               → v:loc{TypeOf v=PT p t}
```

Values of type tobj are triples TO p t o. Its third component o is typed as an obj (that is, a map from strings to properties) refined with a formula stating, for example, that if the object is in the Ref heap, then it contains the "ref" field, and that it satisfies the invariant p. A similar, but more complex invariant applies to objects in the Inv compartment (due to object initialization and implicit effects). The result type of TO records both the invariant p and the tag of the object in the refinement. The type loc is also a triple, TL p t r, where r is a reference to a tagged object. We memoize the tag and invariant of the content of r with the reference itself, and the type of TL ensures that the tags of the content and the loc agree.

Translation of types A source type t in f⋆ is translated to the refined js⋆ type $[\![t]\!] \triangleq$ d:dyn$\{\phi_t\ d\}$ where ϕ_t is a predicate on dyn-typed js⋆ values. We show a few cases below, where SelObj selects an object from the heap, and SelProp selects a property.

$$\begin{aligned}
\phi_{\mathsf{string}} &= \lambda d.\,\mathsf{TypeOf\ d{=}string}\\
\phi_{\mathsf{ref}\ t} &= \lambda d.\,\mathsf{TypeOf\ d{=}PT}\ (\lambda o.\ \phi_t\ (\mathsf{SelProp\ o\ "ref"}))\ \mathsf{Ref}\\
\phi_{t\to t'} &= \lambda d.\,\mathsf{TypeOf\ d{=}WP}\ \lambda_\ a\ _.\Lambda p.\lambda h.\phi_t\ (\mathsf{SelProp(SelObj\ h\ a)\ "0"})\\
&\qquad \wedge \ldots \wedge\ \forall r\ h'.\ \mathsf{Result\ r}\ \phi_{t'} \Longrightarrow p\ r\ h'
\end{aligned}$$

The translation for primitive types like string is simple: the refinement formula ϕ_{string} requires the translated value to be a Str _ : dyn. The translation of ref t requires an object in the Ref heap, whose "ref" field satisfies predicate ϕ_t. The translation of function types $\phi_{t\to t'}$ requires the compiled value to be a Fun wp _ _ term, where (among other requirements) the predicate transformer wp requires its zeroth argument to satisfy ϕ_t, and requires proving the post-condition on a result r that satisfies $\phi_{t'}$ (if it is a value).

Interface of JSVerify To enforce our invariants, JSVerify exposes a monadic version of the JSExec interface. To operate on un values,

for instance, it provides aliases to the functions select, update, and apply of §5. An un value is either a primitive value, or an object (or function) allocated in the Un heap, or a declassified Stub object. The function selectUn allows a (non-internal) field to be selected from an un object. Its pre-condition requires both the caller and the object o to be un-values, and requires the post-condition p to be proven for any IsUn result and heap (since, via getters, selecting a field can trigger arbitrary code). The specification for updating an un object is similar. Calling an un function requires that the caller be an un value, both this and args be un, and ensures that the result is also un. In all cases, the use of the iDST monad requires and ensures the heap invariant as well.

```
type IsUn x = TypeOf x=string ∨ ... ∨ GetTag x = Un
            ∨ (GetTag x = Stub ∧ Declassified x)
type un = x:dyn{IsUn x}
val selectUn:caller:dyn{GetTag caller = Un}
        →o:un →f:string{¬ IsInternalField f}
        →iDST dyn (Λp.λh. ∀r h'. Result r IsUn ⟹ p r h')
val updateUn: caller:dyn{GetTag caller = Un}
        →o:un →f:string{¬ IsInternalField f} →v:un
        →iDST dyn (Λp.λh. ∀r h'. Result r IsUn ⟹ p r h')
val applyUnUn: caller:un →callee:dyn{GetTag callee = Un}
        →this:un →args:un{GetTag args = Un}
        →iDST dyn (Λp.λh. ∀r h'. Result r IsUn ⟹ p r h')
```

Accessing the other heaps imposes stricter requirements but also provides more guarantees.

Relating JSExec and JSVerify To relate our two interfaces, we prove a lemma that shows that any js⋆ program well-typed against JSExec is also well-typed against JSVerify, as long as it interacts only with the Un-fragment of JSVerify. To state this lemma, and in the rest of the paper, we use the following syntactic shorthands:

- We write $S;\Gamma \vdash^D e : t$ for a computation with a trivial pre-condition returning a t-result, i.e., $S;\Gamma \vdash^D e$: iDST t $\mathsf{wp}_\top$, where $S;\Gamma \models \forall$h.HeapInv h$\Longrightarrow \mathsf{wp}_\top(\lambda$_h'.HeapInv h' $\wedge$ DeltaHeap h h')h.
- $S;\Gamma \vdash^D H \mid e : t$ stands for $S;\Gamma \vdash^D H \mid e$: iDST t wp, where $S;\Gamma \models$ wp $(\lambda$_h'.HeapInv h' $\wedge$ DeltaHeap h h') h and h $= asHeap\ H$.
- Γ^D is the lifting of function types in the context Γ, where a type t →t' is lifted to x:t →iDST t' $\mathsf{wp}_\top$. When it is clear from the context, we write types like t →t', leaving the lifting implicit.
- $tagUn(H \mid e)$ is the runtime state obtained by adding Un-tags to each object and loc-typed constant in $H \mid e$.

Lemma 2 (Universal monadic typability of js⋆)**.** *If JSExec*;$\Gamma \vdash H \mid e : t$*, then JSVerify*;$\Gamma^D \vdash^D tagUn(H \mid e) : t$

Henceforth, we write $\Gamma \vdash^D H \mid e : t$ for monadic typing, leaving the signature of JSVerify in the context implicit.

Formal light translation of f⋆ runtime states We now formalize the light translation as a relation $\Gamma \vdash_f H \mid e : t \rightsquigarrow_I H' \mid e'$ for the translation of the f⋆ runtime configuration $H \mid e$ of type t into a js⋆ configuration $I, H' \mid e'$, where H' is the Ref compartment and I consists of both the Inv and Abs compartments. The subscript f is a js⋆ value, representing the object of the function that encloses e; it is Null at the top-level. Figure 5 gives five representative translation rules, simplifying them by eliding type arguments. The rules are to be interpreted as inlining the definitions from JSVerify into the translated term, rather than leaving them as free variables.

The first rule translates a data constructor to an object literal, which in JSVerify is represented as an Inv location that is allocated and immediately initialized with the contents of the constructor. This is particularly important—the alternative of allocating an object first, and then setting its fields is not secure, since, in general, this could cause a traversal of the prototype chain, triggering attacker code in case the attacker has installed a setter on the publicly

$$\frac{\forall i.\Gamma \vdash_f v_i : t_i \rightsquigarrow_I e_i}{\Gamma \vdash_f D_{\bar{t}\to T}\ \bar{v} : T \rightsquigarrow_I \mathsf{mkInv}\ [(\texttt{"tag"},\mathsf{Str}\ (str\ D)));\overline{(str\ i, e_i)}]}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash_f x : t \rightsquigarrow_I \mathsf{selectInv}\ f\ x\ \texttt{"0"}}$$

$$\frac{\Gamma, x{:}t_x \vdash_o e : t \rightsquigarrow_I e' \quad \overline{x_i : t_i} = \mathrm{locals}(e) \quad src \text{ any string constant}}{\Gamma \vdash_f \lambda x{:}t_x.e{:}t_x \to t \rightsquigarrow_I \mathsf{mkFunAbs}\ src\ \lambda o\ \_x.\mathbf{let}\ \overline{x_i = \mathsf{mkLocInv}()}\ \mathbf{in}\ e'}$$

$$\frac{\Gamma \vdash_f e : t' \to t \rightsquigarrow_I e' \qquad \Gamma \vdash_f v : t' \rightsquigarrow_I v'}{\Gamma \vdash_f e\ v : t \rightsquigarrow_I \mathsf{applyAnyAbs}\ f\ e'\ \ell_{global}\ (\mathsf{mkInv}\ (\mathsf{TO}\ \mathsf{Inv}\ [(\texttt{"0"},\mathsf{v'})]))}$$

$$\frac{v_\ell = \mathsf{Obj}\ (\mathsf{TL}\ \mathsf{Inv}\ \ell) \qquad \Gamma \vdash_f v : t \rightsquigarrow_I v' \qquad I(l) = \mathsf{TO}\ \mathsf{Inv}\ [(\texttt{"0"},v')]}{\Gamma \vdash_f v : t \rightsquigarrow_I \mathsf{selectInv}\ \varphi_t\ f\ v_\ell\ \texttt{"0"}}$$

Figure 5. Light translation from f⋆ to js⋆ (selected rules)

available Object.prototype. In contrast, the allocation of an object literal never causes a prototype traversal.

The second rule translates a let- or λ-bound source variable x to a js⋆ expression that selects from the "0" field of an object stored in the Inv heap, whose location is bound to a js⋆ variable of the same name. The invariant guarantees that the "0" field is set in the immediate object, again preventing any prototype traversal.

The third rule translates a closure. Function objects in the light translation are always allocated in the Abs heap. So, we use an alias of JSExec.mkFun from JSVerify called mkFunAbs, which builds the function object. We translate the body of the function using the variable o as the caller object, passed as an argument to applyAbs (an alias of JSExec.apply for calling an Abs function) at every call-site in the body of e, as shown in the next rule. Again, the arguments are passed as an object literal.

The last rule is useful primarily for translating runtime expressions, rather than source values. f⋆ has an applicative semantics with a standard β-reduction rule. However, in js⋆, values are passed as pointers to the Inv (or sometimes Abs) heap. Without this last rule, this mismatch would cause considerable technical difficulties in our forward simulation proof. For example, in f⋆ we may have $(\lambda x.(x,x))\ D \to_S (D,D)$ for some constructor D. When translating the left-hand side, we may allocate only one D in js⋆, whereas, the translation of the right-hand side would allocate two objects. To reflect both possibilities, the light translation is a non-deterministic relation on runtime states, indexed by the js⋆ heap I, representing pre-allocated data. So, in the last rule, if we find a location ℓ in the I heap which already contains a value that is a valid translation of the source value v, then, rather than allocate a fresh location, we may simply translate v to the expression that selects from ℓ. As such, our translation relation conveniently hides the details of data allocation and aliasing—our typed invariant and §8 show that those details are not observable anyway.

Correctness of the light translation We present our main results for the light translation, first stating that it preserves the typing and heap invariants, then that it is a forward simulation: every f⋆ reduction is matched by one or more js⋆ reductions.

Type preservation states that if an f⋆ state $H \mid e$ well-typed at t is translated to a js⋆ state $H' \mid e'$ (with Inv and Abs heaps I), then the js⋆ state is well-typed in the iDST monad against JSVerify. The lemma ensures that when h0 (the logical value corresponding to I and H') satisfies the heap invariant, the js⋆ state diverges, or produces a result r and post-heap h1, where h1 satisfies the heap invariant and DeltaHeap h0 h1; and that Result r ϕ_t is valid. As a base case, the heap invariant on the empty heap (produced when translating a source program, rather than an intermediate runtime configuration) is trivially satisfied. A technical requirement, due to JavaScript's

hoisting of local variables, is that all the let-bound variables of the translated term already exist in the heap I.

Lemma 3 (Type preservation)**.** *If* $\Gamma \vdash_f H \mid e : t \rightsquigarrow_I H' \mid e'$ *then, for* $\Gamma' = [\![\Gamma]\!], \Gamma_{locals(e)}$*, there exists* ψ *such that* $\Gamma' \vdash^D H', I \mid e' :$ *iDST dyn* ψ *and, for* $h0 = asHeap\ (I, H')$*, if HeapInv h0 and LocalsOK locals*(e) *h0, then*

$$\Gamma' \models \psi\ (\lambda x\ h1.\ HeapInv\ h1 \wedge DeltaHeap\ h0\ h1 \wedge Result\ x\ \phi_t)\ h0$$

Our next lemma ensures that the formal light translation is a forward simulation. That is, every reduction step of an f* program $H \mid e$ is matched by one or more reductions of its js* translation. We use an auxiliary function *Abs I*, standing for the set of objects that may be used as the caller object in js*:

$$Abs\ I = \{\mathsf{Null}\} \cup \{\mathsf{Obj}\ \ell \mid \ell \in \mathrm{dom}(I) \wedge \mathsf{GetTag}\ \ell = \mathsf{Abs}\}$$

Lemma 4 (Forward simulation)**.** *For any source reduction step* $H \mid e \rightarrow_S H_1 \mid e_1$ *and any translation* $\Gamma \vdash_f H \mid e \rightsquigarrow_I H' \mid e'$*, where* $f \in Abs\ I$*, there exist reduction steps* $I, H' \mid e' \rightarrow_S{}^+ I, I', H'_1 \mid e'_1$*, and a translation* $\Gamma \vdash_g H_1 \mid e_1 \rightsquigarrow_{I,I'} H'_1 \mid e'_1$*, where* $g \in Abs\ (I, I')$*.*

Defensive wrappers We now consider the properties of the second phase of our compiler, i.e., the defensive wrappers. Figure 6 lists the js* code of `downfun` and `upfun`. It is instructive to compare with the JavaScript wrappers shown in Figure 4—we discuss below a few simplifications to the code in support of verification. This code is typed against JSVerify, making use of the heap-partition-aware variants of functions in JSExec. This allows us to record the code positions that may trigger callbacks to untrusted code (which leaks the caller's object to the context). Specifically, we use the following variants of mkFun, apply, select, and update.

- abs-functions reside in the Abs heap and are created by mkFunAbs. They may be called with any caller using applyAnyAbs.
- un-functions are created by mkFunUn with an object o:un. They are called using applyUnUn, that is, only by callers with an un object (since this object is possibly leaked to the callee) and un arguments. Similarly, Un objects are selected using selectUn, possibly triggering a callback to the context (due to a getter).
- stub-functions specifically support our 'upfun' wrapper. They are created by mkFunStub with an object in the Stub compartment. They can be called *at most once* by any caller using applyAnyStub, after which they are declassified and released to the context. Prior to the declassification, stub objects are safe—they can be updated without triggering callbacks.
- Local variables and data constructors are allocated in the Inv heap using mkLocalInv and mkInv, respectively. These local variables may be set at most once, using setInv, then selected many times using selectInv. These calls never trigger callbacks.
- Mutable references are allocated using mkRef, and accessed using selectRef and updateRef. Locations in the Ref compartment are always well-typed, and the accesses never trigger callbacks.

For a given source type t, $\downarrow t$ is an abs-function that takes values of type $[\![t]\!]$ and returns values of type un. Conversely, $\uparrow t$ is an un-function that takes values of type un and (attempts to) return a value of type $[\![t]\!]$. To facilitate proofs of these typing properties, we instrument the js* wrappers with calls to JSVerify, rather than JSExec, as already noted. Additionally, we require four verification hints in the code of upfun. First, we add a call to a ghost function declassify u callee, which is used to record in the refinement logic that the stub object has been released to the attacker and should henceforth be typed as un. A pre-condition of declassify u callee is that the all the fields of callee must already by typeable as un. Hence, we set the `"caller"` field to the stub itself, simulating the

```
let downfun =
  mkFunAbs "downfun" (fun _ _ (apair:inv) →
    mkFunAbs "downfun_a2b" (fun _ _ (af:inv) →
      mkFunUn "downfun_f" (fun (u:un) (_:un) (az:un) →
        let up_a = selectInv u apair "0" in
        let z = selectUn u az "0" in
        let x = applyUnUn u up_a global (mkArgUn z) in
        let f = selectInv u af "0" in
        let y = applyAnyAbs u f global (mkArgInv x) in
        let down_b = selectInv u apair "1" in
        applyAnyAbs u down_b global (mkArgInv y))))
let upfun =
  mkFunAbs "upfun" (fun _ _ (apair:inv) →
    mkFunUn "upfun_a2b" (fun _ _ (af:un) →
      mkFunAbs "upfun_f" (fun (o:abs) _ (ax:inv) →
        let az = mkLocalInv() in
        let by = mkRef (mkInv [("tag", Str "None")]) in
        let down_a = selectInv apair "0" in
        let x = selectInv o ax "0" in
        setInv o az "0" (applyAnyAbs o down_a global (mkArgInv x));
        let stub = mkFunStub "stub" (fun (u:stub) (_:un) (a0:stub) →
          let callee = selectStub u a0 "callee" in
          updateStub u callee "caller" callee;
          declassify u a0; (* ghost *)
          declassify u callee; (* ghost *)
          let f = selectUn u af "0" in
          let z = selectInv u az "0" in
          let y = applyUnUn u f global (mkArgUn(z)) in
          let up_b = selectInv u apair "1" in
          let b = applyUnUn u up_b global (mkArgUn y) in
          let someb = mkInv [("tag", Str "Some"); ("0", b)] in
          updateRef u by "ref" someb)
        applyAnyStub o stub global (allocStub());
        selectInv o (selectRef o by "ref") "0")))
```

Figure 6. Function wrappers in js* (omitting most types)

recursive call in the upfun wrapper of Figure 4 (which, experimentally, we confirm has the behavior of introducing a cycle in the `caller` chain).[1] As a third hint we add a ghost call to declassify u a0. Since a0 is also a field of callee, we will release a0 to the attacker when the call to f proceeds. Since it is a reference to an empty object in the stub heap (or an object with a single constant boolean field in Figure 4), this does not leak any information to the attacker, and so it can be declassified trivially. A further complication is that the callee object has an internal field called `"@code"` containing a reference to the function closure itself, which the adversary can use to call the stub directly, once it has access to the stub function object (e.g., by using `Function.prototype.apply`). JavaScript provides no way to clear the `"@code"` field directly. To handle this case, we carefully ensure that, after declassification, the function closure can be typed as a function from un to un. Thus, the stub returns its result via a side-effect to the reference by. Typing this idiom requires one level of indirection (our final hint): we initialize the reference by to None and, each time the stub is called and successfully imports the translation of a source value v:b, it updates by with the translation of Some v. In Figure 4, we collapse the option reference into a single mutable location, which is a simple semantics-preserving transformation.

Equipped with these types, we show that a down-wrapped light translation has type un. Likewise, we show that, if an up-wrapped un value returns normally, then it returns a value typed as the translation of its source type. In the lemma statement, we write $\downarrow t\ e$ for the application of a down wrapper to e, i.e., applyAnyAbs $(\_ : abs)\ \downarrow t\ (\_ : un)\ e$; and $\uparrow t\ e$ is applyUnUn $(\_ : un)\ \uparrow t\ (\_ : un)\ e$. In

[1] We anticipate enhancing our proof of wrapper typing and the bisimulation of §7 to account directly for the recursive call in `stub`, instead of using the assignment to `"caller"`.

conjunction with Lemma 2, this shows that a wrapped term can be safely embedded in any JavaScript context.

Lemma 5 (Typing of wrapped terms)**.**
If $\Gamma \vdash^D v : [\![t]\!]$ *then* $\Gamma \vdash^D \downarrow t\ v : un$*; if* $\Gamma \vdash^D v : un$ *then* $\Gamma \vdash^D \uparrow t\ v : [\![t]\!]$.

7. Contextual equivalence by bisimulation in $\mathsf{f}^\star$

Contextual equivalence is a precise and intuitive notion of equivalence, both in JavaScript and in $\mathsf{f}^\star$, but it leads to complicated direct proofs, as one needs to reason about any reduction in any context. To structure our full-abstraction proof, and to analyze interactions between translations of equivalent $\mathsf{f}^\star$ expressions and their $\mathsf{js}^\star$ contexts, we develop a custom labeled bisimulation proof technique. Although formally independent of JavaScript, the design of our bisimulation is guided by its application to source $\mathsf{f}^\star$ and $\mathsf{js}^\star$ in §8:

- Our bisimulation must support $\mathsf{f}^\star$ types, higher order functions, mutable state, exceptions, divergence, and errors.
- Functions exported using 'down' wrappers may share private state, so we need to jointly relate *configurations* of functions, rather than single functions. (See also Sumii and Pierce 2005.)
- Our wrappers stop at imported and exported functions; thus, to extend wrapping from terms to configurations and maintain wrapping as a transition invariant, we use a variant of normal form bisimulation (Lassen 2005): our configurations have free variables standing for the functions imported from the context; thus, any callback yields a transition output with a continuation.

Next, we define these bisimulation configurations and we study their behavior, first using concrete context closures, then more abstract labeled transitions. The main result of the section is that labeled bisimilarity coincides with contextual equivalence.

We use interfaces to specify how configurations may interact with their context. An interface declares some exported functions (previously sent to the context), some imported functions (previously received from the context), some continuations (for ongoing calls to the context), and some memory (shared with the context).

Definition 6 (Interface)**.** *An interface* $\mathscr{I} = \sigma;\Gamma$ *consists of heap types* σ *(for the heap shared with the context) and a type environment* Γ *that binds variables to function types* t*, each annotated with one of three sorts:*

- $z :_{\mathsf{z}} t$ *for functions imported from the context;*
- $x :_{\mathsf{x}} t$ *for functions exported to the context; and*
- $k :_{\mathsf{k}} t$ *for continuations of calls to the context.*

We let Γ_{z} *be the projection of* Γ *on imported functions, and similarly for* Γ_{x} *and* Γ_{k}*. We often elide these annotations, writing for instance* $y : t \in \Gamma$ *for any binding of* Γ*. When* $y \in \mathrm{dom}(\Gamma)$*, we write* $\Gamma|y$ *for the prefix of* Γ *such that* Γ *is of the form* $\Gamma|y, y{:}t, \Gamma'$.

Our configurations represent *pairs* of related $\mathsf{f}^\star$ runtime states, both waiting for their next interaction with the context. We introduce notations for pairs of related terms in configurations: for any phrase of syntax M, we write $\overline{M}$ for pairs of Ms, and write $M_\urcorner$ and $M_\ulcorner$ for their left and right projections, respectively. Further, we treat propositions φ with pairs $\overline{M}^n_{i=1}$ as $\varphi\{M^i_\urcorner/\overline{M}^i\} \wedge \varphi\{M^i_\ulcorner/\overline{M}^i\}$.

In this section and §8, we omit the inductive signature S in typing judgments, and write ';' instead of ',' to separate references from functions in typing environments.

Definition 7 (Configuration)**.** *Given an interface* $\mathscr{I} = \sigma;\Gamma$*, a well-formed configuration* $\mathscr{C} = \bar{I}\,|\,\overline{\rho}$ *(written* $\mathscr{I} \vdash \mathscr{C}$*) consists of two heaps* $\bar{I}$ *such that* $\sigma;\Gamma_{\mathsf{z}} \vdash \bar{I}$ *and two substitutions* $\overline{\rho}$ *from every* $y{:}t \in \Gamma_{\mathsf{x}} \uplus \Gamma_{\mathsf{k}}$ *to values* $\overline{v}$ *such that* $\sigma, \sigma(\bar{I}); (\Gamma|y)_{\mathsf{z}} \vdash \overline{v} : t$.

In the definition, $\bar{I}$ is a pair of private heaps, with $I_\urcorner$ and $I_\ulcorner$ having possibly different domains, both disjoint from σ. (The typing judgments imply $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma(I)) = \emptyset$.) The substitutions $\overline{\rho}$ map every variable in $\Gamma_{\mathsf{x}} \uplus \Gamma_{\mathsf{k}}$ to functions; in particular, continuations are just functions. The environment $(\Gamma|y)_{\mathsf{z}}$ let us type them with free variables standing for previously-imported functions.

Next, we lift the contextual equivalence of §3 from terms to configurations, relying on generalized contexts and context closure:

Definition 8 (Configuration Context)**.** *Given an interface* $\sigma;\Gamma$*, a well-formed evaluation context* $\mathscr{E} = O\,|\,Z, e : t$ *consists of*

- *a heap* O *such that* $\sigma \subseteq \sigma(O)$ *and* $\sigma(O);\Gamma_{\mathsf{x}} \vdash O$*;*
- *a substitution* Z *from every* $z{:}t \in \Gamma_{\mathsf{z}}$ *to a value* v *such that* $\sigma(O); (\Gamma|z)_{\mathsf{x}} \vdash v : t$
- *a call-stack: a typed expression* $e : t$ *defined by induction on* Γ_{k}*:*
  - *when* $\Gamma_{\mathsf{k}} = \emptyset$*, the stack is any* e *such that* $\sigma(O);\Gamma_{\mathsf{x}} \vdash e : t$*;*
  - *when* $\Gamma_{\mathsf{k}} = k : t_1 \to t_2, \Gamma'_{\mathsf{k}}$*, the stack is any* $E[k\ e_1]$ *such that* $\sigma(O); (\Gamma|k)_{\mathsf{x}}, y : t_2 \vdash E[y] : t$ *and* $e_1 : t_1$ *is a stack for* Γ'_{k}.

Definition 9 (Context Closure)**.** *Given a context* $O\,|\,Z, e : t$ *and a configuration* $\bar{I}\,|\,\overline{\rho}$ *with the same interface and disjoint heaps (*$\mathrm{dom}(O) \cap \mathrm{dom}(\bar{I}) = \emptyset$*), we let* $\mathscr{E}[\mathscr{C}] \triangleq (O, \bar{I} \mid e)\{\overline{\rho}, Z\}$ *be the two runtime states obtained by composing the context with both sides of the configuration and jointly applying the substitutions* $\overline{\rho}$ *and* Z *in the order recorded in* Γ *(substituting first the latest variables).*

These runtime states are closed and well-typed: $\vdash \mathscr{E}[\mathscr{C}] : t$

Definition 10 (Contextual equivalence for Configurations)**.**

$$\approx_E \triangleq \{\mathscr{I} \vdash \mathscr{C} \text{ such that } \forall \mathscr{E}, \mathscr{E}[\mathscr{C}]_\urcorner \approx^\bullet_{\mathsf{e}} \mathscr{E}[\mathscr{C}]_\ulcorner\}$$

Our definition generalizes plain contextual equivalences: for instance, contextual equivalence $\approx_{\mathsf{e}}$ on functions of type t coincides with $\approx_E$ on configurations with signature $\emptyset; x :_{\mathsf{x}} t$. More generally, we can reduce contextual equivalence of two open expressions $x_i : t_i \vdash \overline{e} : t$ to the function-value equivalence of $\vdash \lambda x_i.\overline{e} : t_i \to t$.

Labeled bisimulations To keep the context implicit, we define interactions between the configuration and its context as labeled transitions of the form $\mathscr{I} \vdash \mathscr{C} \xrightarrow{\alpha;\beta} \mathscr{I}' \vdash \mathscr{C}'$ where α and β range over input and output labels, respectively. The input label (e.g., a function call with parameters, or returning a result to a previous call) is provided by the context to the configuration, and the output label is its response (e.g., a returned value or a callback). The transition occurs only if both terms in the configuration $\mathscr{C}$ have matching behaviors (e.g., both return the same result, call the same callback, etc). Thus, the transition relation characterizes pairs of $\mathsf{f}^\star$ runtime states that have similar interactions with their context.

Definition 11 (Input Label)**.** *Given an interface* $\sigma;\Gamma$*, an input label* $\alpha = \Gamma'_{\mathsf{z}}\ O\,|\,y\,r$ *consists of*

- *a signature* Γ'_{z} *disjoint from* Γ*;*
- *a heap* O *such that* $\sigma \subseteq \sigma(O)$ *and* $\sigma(O);\Gamma_{\mathsf{x}}, \Gamma'_{\mathsf{z}} \vdash O$*;*
- *a value parameter* v *such that* $\sigma(O);\Gamma'_{\mathsf{z}} \vdash v : t$*;*
- *a query* $q = y\,r$*, of one of the three forms below:*
  - $x\,r$*, a call to any* $x : t \to t' \in \Gamma_{\mathsf{x}}$*; or*
  - $k\,v$*, a return with* $\Gamma_{\mathsf{k}} = \Gamma'_{\mathsf{k}}, k : t \to t'$*; or*
  - $k\,(\textbf{\textit{raise}}\ v)$*, an exception with* $\Gamma_{\mathsf{k}} = \Gamma'_{\mathsf{k}}, k : t'$ *and* $t = t_e$.

such that, moreover, the values within O *and* v *with a function type are pairwise distinct variables in* Γ'_{z}.

Intuitively, an input label represents a minimal, open context that calls a function previously exported by the configuration, or returns from a previous call from the configuration to the context; O represents some shared heap, and Γ'_{z} some fresh variables standing for any function imported by this input from the context. In combination, we obtain a pair of well-typed (open) runtime states $\Gamma_{\mathsf{z}}, \Gamma'_{\mathsf{z}} \vdash O, \bar{I} \mid y\,\overline{\rho}\,v$ applying the function associated with y in $\overline{\rho}$

to v. In the last two forms of query, the condition on Γ_k ensures that the context returns only at the top of the stack (with a value or an exception). The final condition ensures that no function values are passed directly to the configuration. In particular, the context cannot directly pass a function x previously received from the configuration. Instead, the context can pass a fresh variable z, and can later call x whenever z is called back.

Definition 12 (Output Label)**.** *Given an interface $\sigma;\Gamma$ and an input label $\Gamma'_z\ O \mid y\ r : t$, an output label is one of the following:*

- ***error** for failure;*
- *$\Uparrow$ for divergence;*
- *$\Gamma''_x\ O' \mid v'$ for normal returns, such that $\Gamma''_x \vdash O' \mid v' : t$;*
- *$\Gamma''_x\ O' \mid$ **raise** v' for exceptions, such that $\Gamma''_x \vdash O' \mid v' : t_e$; or*
- *$\Gamma''_x\ O' \mid z\ v' : t''$ for callbacks, such that $\Gamma''_x \vdash O' \mid v' : t'$ and $z : t' \to t'' \in \Gamma'_z$.*

In the last three cases, we require that Γ''_x and Γ' be disjoint; that O' extend O with the locations reachable from v' and O; and that all values within O' and v' with function types be distinct variables defining the domain of Γ''_x.

For convenience, to deal with both kinds of inputs, we write $\Gamma \setminus_k y$ for Γ when $y :_x \_ \in \Gamma$ and for $\Gamma \setminus y$ when $\Gamma_k = \Gamma'_k, y : \_$. We also let Γ^+ abbreviate $\Gamma \setminus_z y, \Gamma'_z, \Gamma''_x$ in the interface after the output. As with input labels, we require that v' in output results do not contain names from Γ_x, Γ_z.

In combination, compatible inputs and outputs define transitions between interfaces. Transitions between configurations (defined shortly) can be seen as their refinement, of the form

$$\sigma;\Gamma \vdash \mathscr{C} \xrightarrow{\Gamma'_z\ O \mid q\ v:t\ ;\ \Gamma''_x\ O' \mid z\ v':t''} \sigma(O');\Gamma^+ \vdash \mathscr{C}'$$

A plain labeled simulation on pair of terms would require that, for every transition on one side, there exists a matching transition on the other side. In our case, since configurations already account for both sides, we define transitions as a *partial* relation between configurations: we have a transition only when both sides perform matching outputs. For uniformity, we use two additional terminal configurations, **error** and $\Uparrow$, with no input, to represent the outcome of transitions that lead to failure and divergence, respectively.

For a given configuration $\mathscr{I} \vdash \mathscr{C}$, the additional locations in O in the input label may clash with the private locations of $\overline{I}$; our transitions implicitly assume that this is not the case, as we can always pick another input label with fresh names, and that the output label is well-formed.

Definition 13 (Transition)**.** *Given interface $\sigma;\Gamma$ and input label $\alpha = \Gamma'_z\ O \mid y\ r : t$, configuration transitions are (partially) defined below, with one rule for each form of output:*

$$\frac{O\rho_b, I_b \mid y\rho_b\ v \Downarrow \textbf{\textit{error}} \text{ for } b = \neg, \Rsh}{\sigma;\Gamma \vdash \overline{I} \mid \overline{\rho} \xrightarrow{\alpha\ ;\ \textbf{\textit{error}}} \textbf{\textit{error}}} \qquad \frac{O\rho_b, I_b \mid y\rho_b\ v \Uparrow \text{ for } b = \neg, \Rsh}{\sigma;\Gamma \vdash \overline{I} \mid \overline{\rho} \xrightarrow{\alpha\ ;\ \Uparrow} \Uparrow}$$

$$\frac{\begin{array}{c} O\rho_b, I_b \mid y\rho_b\ v \to_S^* O'\rho'_b, I^\circ{}_b \mid r'\rho'_b \text{ for } b = \neg, \Rsh \\ \sigma;\Gamma \vdash \overline{I} \mid \overline{\rho} \xrightarrow{\alpha\ ;\ \Gamma''_x\ O' \mid r'} \sigma, \sigma';\Gamma^+ \vdash \overline{I'} \mid \overline{\rho}, \overline{\rho}' \\ O\rho_b, I_b \mid y\rho_b\ v \to_S^* O'\rho'_b, I^\circ{}_b \mid E_b[z\ v'\rho'_b] \text{ for } b = \neg, \Rsh \end{array}}{\sigma;\Gamma \vdash \overline{I} \mid \overline{\rho} \xrightarrow{\alpha\ ;\ \Gamma''_x\ O' \mid z\ v'} \sigma, \sigma';\Gamma^+, k :_x t \vdash \overline{I'} \mid \overline{\rho}, k \mapsto \lambda r.\overline{E}[r], \overline{\rho}'}$$

Definition 14 (Bisimilarity)**.** *A bisimulation φ is a set of configurations closed by transitions: for every $\mathscr{I} \vdash \mathscr{C} \in \varphi$ with input label α, there exists an output label β and $\mathscr{I}' \vdash \mathscr{C}' \in \varphi$ such that*

$$\mathscr{I} \vdash \mathscr{C} \xrightarrow{\alpha\ ;\ \beta} \mathscr{I}' \vdash \mathscr{C}'$$

Applicative bisimilarity, written $\approx_A$, is the largest bisimulation.

As usual, to prove $\mathscr{C} \subseteq \approx_A$ for a given configuration $\mathscr{C}$, it suffices to build some φ that includes $\mathscr{C}$ and show that it is a bisimulation. (See the full paper for additional discussion and examples.)

We now relate labeled transitions on configurations to the reductions of runtime states obtained by context closure.

Lemma 15. *Let $\mathscr{C} \in \approx_A$. The two sides of any context closure $\mathscr{E}[\mathscr{C}]$ either return some identical result; or diverge; or reduce to some $\mathscr{E}'[\mathscr{C}']$ with $\mathscr{C} \xrightarrow{\alpha\ ;\ \beta} \mathscr{C}'$ and $\mathscr{C}' \in \approx_A$.*

Lemma 16. *For every transition $\mathscr{I} \xrightarrow{\alpha\ ;\ \beta} \mathscr{I}'$, there is a context $\mathscr{E}_T$ for $\mathscr{I}$ such that, for any $\mathscr{C} : \mathscr{I}$ with input α, we have $\mathscr{E}_T[\mathscr{C}] \in \approx_e^\bullet$ if and only if $\mathscr{C}$ performs this transition.*

Lemma 17. *For every transition $\mathscr{I} \vdash \mathscr{C} \xrightarrow{\alpha\ ;\ \beta} \mathscr{I}' \vdash \mathscr{C}'$ and context-closure $\mathscr{E}'[\mathscr{C}']$, there is a context closure $\mathscr{E}[\mathscr{C}]$ such that $\mathscr{E}[\mathscr{C}] \to_S{}^* \approx_e^\bullet \mathscr{E}'[\mathscr{C}']$.*

Theorem 4. $\approx_E = \approx_A$.

8. Full abstraction for wrapped translations

We are finally ready to prove full abstraction, relating contextual equivalences in $\mathsf{f}^\star$ and $\mathsf{js}^\star$. Relying on Theorem 4, we use labeled bisimulations rather than contextual equivalences: the main idea is to extend the wrapped translation from source expressions to source configurations, and to systematically relate their source and target transitions. We begin by establishing a corollary of type soundness for reasoning about untrusted callbacks. Then, we establish operational properties for wrappers, setting up notations for configuration translations (Definition 21).

Untrusted callbacks As we import untrusted values, callbacks triggered by getters are innocuous, since the context can provide arbitrary values anyway. To deal with them in the bisimulation, we use a corollary of open subject reduction, where we use I to refer to the union of the translation of the source Ref heap H, together with the Inv, Abs, and the private locations in the Stub heap.

Lemma 18 (Untrusted Callbacks)**.** *For every typed open runtime state $s \triangleq O_{un}, I \mid e$, where $\Gamma'_z \vdash^D s : t'$ and Γ'_z defines only functions $z : un \to un$, one of the following holds: $s \Uparrow$; or $s \to_S{}^* O'_{un}, I^+ \mid r$; or $s \to_S{}^* O'_{un}, I^+ \mid E[z\ (\_ : un) : un]$.*

Properties of wrappers For functional types, the full paper defines $\downarrow t\{x'\}$, the $\mathsf{js}^\star$ open function value obtained by reducing $\downarrow t\ x'$. Similarly, $\uparrow t\{z\}$ is the function obtained by reducing $\uparrow t\ z$—this latter operation just distributes up and down wrappers around the argument and return of z, so the value always exists. In both cases, we remove the Fun constructor, and retain the function closure within. The main operational property for down wrappers is that, as we export a translation of a source value, we obtain a freshly-allocated un value that depends only on the source value—not on the choice of its translation. Its proof is by induction on v relying on monadic typing (§6) and is mutually inductive with the proof of Lemma 20, which reasons about importing values from the context. For clarity, we present the two lemmas separately.

Lemma 19 (Running down)**.** *Let $\Gamma_x \vdash v : t$ such that $\downarrow t$ is defined and Γ_x binds the functions of v. Let Γ'_x declare $x : un \to un \to un$ and ρ substitute $\downarrow t'\{x\}$ for every $x :_x t' \in \Gamma_x$. There exist O_{un} and v' such that, for any light translation $\Gamma_x \vdash_f v : t \rightsquigarrow_I e'$, we have $I \mid \downarrow t\ e' \to_S{}^* O_{un}, I^+ \mid v'\rho$ and $\Gamma'_x \vdash^D O_{un}, I^+ \mid v'\rho : un$.*

Intuitively, O_{un} is the new un memory allocated to copy the exported value, while Γ'_x declares the exported wrapped-down translations of the function variables in v.

The next lemma shows that 'up' wrappers return (at most) light translations of a source value that depend only on the untrusted

context. Besides, those wrappers may trigger untrusted callbacks in the process, diverge, fail, or raise an exception. (In addition to divergence in any callbacks that may be triggered, we may get divergence, for instance, as we try to import a circular list.)

Lemma 20 (Running up). *For all values $u : \mathit{un}$ and states $s \triangleq O_{un}, H, I \mid \uparrow t\ u$, such that $\Gamma'_{\mathsf{z}} \vdash^{D} s : [\![t]\!]$, if s returns with final state $O'_{un}, H', I^{+} \mid v' : [\![t]\!]$ (after any number of untrusted callbacks) then $\Gamma_{\mathsf{z}} \vdash_{f} H'' \mid v : t \rightsquigarrow_{I^{+}} H' \mid v'$ for some source runtime state $H'' \mid v$.*

Candidate bisimulation We define the full translation of source configurations. (The full translation of programs is a special case.) As in §4, this translation is non-deterministic; it includes consistent translations of every piece of the source configuration. It is designed to be closed by $\mathsf{js}^{\star}$ transitions. To keeps track of all allocated stubs, it also includes the translation of H_{stub}, an auxiliary store (not present in the source configuration) with one option reference for every stub. As can be expected of a defensive translation, its interface consists entirely of untrusted locations and functions.

We refer to the full paper for auxiliary definitions of the evaluation contexts $E_{\downarrow t}[\_]$ for frames that transition from untrusted callers to translated callees, $F[F'[\_]]$ for frames that transition from translated callbacks to untrusted callees, $\mathit{stub}_{s,r}$ for $\mathsf{f}^{\star}$ stub closures with object s and reference r (variable by in Figure 4), and $U[\_]$ that ranges over their continuations after declassification. This definition also omits partially-applied functions, e.g. $\downarrow t\{x\}$ applied to a this object and waiting for its arguments object.

Definition 21 (Configuration Translation). *A translation of the source configuration $\emptyset; \Gamma \vdash \mathscr{C} = \overline{H} \mid \overline{\rho}$ is any configuration of the form $\sigma'; \Gamma' \vdash \overline{I}', \overline{H}' \mid \overline{\rho}'$ such that, for $b = \curvearrowleft, \curvearrowright$ we have*

(1) $\Gamma \vdash H_b \uplus H_{\mathsf{stub}} \rightsquigarrow_{I_b} H_b^{\circ}$.
(2) *For every $x :_{\mathsf{x}} t \in \Gamma$, we have $\Gamma \vdash x\rho_b \rightsquigarrow_{I_b} v_b^{\circ}$ and $x\rho_b^{\circ} = \downarrow t\{v_b^{\circ}\}$.*
(3) *For every $k :_{\mathsf{k}} t \to t' \in \Gamma$, we have $\Gamma \vdash k\rho_b \rightsquigarrow_{I_b} E_b^{\circ}[\_]$, using the light translation of expressions with an additional rule translating $[\_]$ to $[\_]$, and $k\rho_b^{\circ}$ is $E_{\downarrow t'}[E_b^{\circ}[F[F'[U[\_]]]]]$ for some typed continuation $U[\_]$ reachable from $\mathit{stub}_{s,r}$.*
(4) *For every $r \mapsto v : \mathsf{option}\, t' \in H_{\mathsf{stub}}$, ρ_b° also defines $\mathit{stub}_{s,r}$ and any typed continuations $U[\_]$ reachable from $\mathit{stub}_{s,r}$.*
(5) *$\overline{I}'$, $\overline{H}'$, and $\overline{\rho}'$ are obtained from $\overline{I}^{\circ}$, $\overline{H}^{\circ}$, and $\overline{\rho}^{\circ}$ by replacing every instance of z with $\uparrow t\{z\}$ for every $z :_{\mathsf{z}} t \in \Gamma$.*
(6) *σ' declares un objects (including function objects for the exported functions x) and the function objects s of the stubs of* (4).
(7) *Γ' declares $[\![z :_{\mathsf{z}} t]\!] = z :_{\mathsf{z}} \mathit{un} \to \mathit{un} \to \mathit{un}$, $[\![x :_{\mathsf{x}} t]\!] = x :_{\mathsf{x}} \mathit{un} \to \mathit{un} \to \mathit{un}$, or $[\![k :_{\mathsf{k}} t]\!] = k :_{\mathsf{k}} \mathit{un} \to \mathit{un}$ for every declaration in Γ and every definition of* (4)*,*

The lemma statements below account for the non-determinism of our translation: for soundness, $[\![\_]\!]$ collects all configuration translations defined above; for completeness, $[\![\_]\!]'$ collects all configuration translations without stub callbacks. We obtain our main theorem for programs seen as singleton configurations.

Lemma 22 (Soundness). *If $\mathscr{I} \vdash \mathscr{C} \in \approx_E$ then $[\![\mathscr{I} \vdash \mathscr{C}]\!] \subseteq \approx_E$.*

Lemma 23 (Completeness). *If $[\![\mathscr{I} \vdash \mathscr{C}]\!]' \cap \approx_E \neq \emptyset$ then $\mathscr{I} \vdash \mathscr{C} \in \approx_E$.*

Theorem 5 (Full abstraction). *For all translations $\vdash \overline{v} : t \rightsquigarrow_{\emptyset} \overline{e}$ such that $\downarrow t$ is defined, $v_0 \approx_{\mathsf{e}} v_1$ in $\mathsf{f}^{\star}$ if and only if $\downarrow t\ e_0 \approx_{\mathsf{e}} \downarrow t\ e_1$ in $\mathsf{js}^{\star}$.*

9. Preliminary case studies and discussion

Although we leave an extensive evaluation of our compiler as future work, we have already used it to program several small case studies (available on the web)—we briefly describe two of these here.

Secure subsystems A traditional challenge in JavaScript programming involves combining code from multiple, mutually distrusting sources, while maintaining a degree of control over the resulting mash-up. One design towards this objective could be to implement a subsystem in $\mathsf{f}^{\star}$ that mediates mash-up interactions, prove it correct using monadic $\mathsf{f}^{\star}$, and then deploy it using our compiler.

To illustrate our point, we have implemented an interference-free local store on top of the `localStorage` object in HTML 5, which offers a key-value store per web page. The challenge is that this resource is shared between all scripts running in a web page: they can all read, write, and even clear the whole storage without any access control. To enable mash-ups to use local storage, we implement a secure API `setItem`, `getItem`, and `removeItem` that multiplexes access to `localStorage` while ensuring isolation.

Advising JavaScript Meyerovich and Livshits (2010) propose ConScript, a browser-based implementation of *aspect-oriented advice* for JavaScript, itself expressed as JavaScript code. In preliminary experiments, we have been able to implement 6 ConScript policies in $\mathsf{f}^{\star}$. Being written in $\mathsf{f}^{\star}$ makes the advice simpler: we can prove correctness of the advice by contextual equivalence at the source level, and, unlike ConScript, we do not require any browser modifications.

Performance The benefits of running secure JavaScript come at a price, as any interactions with untrusted code are mediated by wrappers. The cost of wrapping, however, is proportionate only to the number of "boundary crossings" between trusted and untrusted code. When executing within $\mathsf{f}^{\star}$, there is little overhead due to security protections. Nevertheless, we expect to improve our translation, with an eye towards performance, along several axes. For example, our current representation of datatypes is naïve. We might instead use JavaScript's `ArrayBuffer`, which offers a packed data representation. For the wrappers, rather than exporting data by copying, we plan to investigate using `Object.freeze`, a new feature that dynamically renders the fields of an object immutable. We are also considering using a lazy semantics for the wrappers that import data (possibly using proxies of the forthcoming ES6 standard). This would necessitate reflecting the callbacks in the source semantics, as lazy importing may trigger callbacks to the context as imported data is read, but the performance gains may make this a good tradeoff.

Conclusions It is increasingly common for compilers to target JavaScript, contributing to a growing view of JavaScript as the "assembly language of the Web". Our work provides a foundation for such compilers, particularly when compiled code must interact with code from other, less trustworthy sources. Relying on full abstraction, developers who program in higher level languages such as ML can reliably and securely deploy their code, without having to worry about the intricacies of JavaScript.

References

M. Abadi. Protection in programming-language translations. In *ICALP*, volume 1443, pages 868–883, 1998.

M. Abadi and G. D. Plotkin. On protection by layout randomization. In *IEEE CSF*, pages 337–351, 2010.

M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, Apr. 2002.

P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *IEEE CSF*, pages 171–185, 2012.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, 2008.

Caja. Attack vectors for privilege escalation, 2012. URL `http://code.google.com/p/google-caja/wiki/AttackVectors`.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, 2006.

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

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

A. Kennedy. Securing the .NET programming model. *TCS*, 364(3), 2006.

S. Lassen. Eager normal form bisimulation. *LICS*, 2005.

S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.

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

L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, 2010.

J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141 – 163, 1993.

J. H. Morris. Protection in programming languages. In *CACM (16)*, 1973.

J. Politz, M. Carroll, B. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *DLS*, 2012.

C. Schlesinger and N. Swamy. Verification condition generation with the Dijkstra state monad. Technical Report MSR-TR-2012-45, Mar. 2012.

M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.

E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *POPL*, 2005.

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

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Towards JavaScript verification with the Dijkstra state monad. Technical Report MSR-TR-2012-37, Mar 2012.

A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE S&P*, 2011.