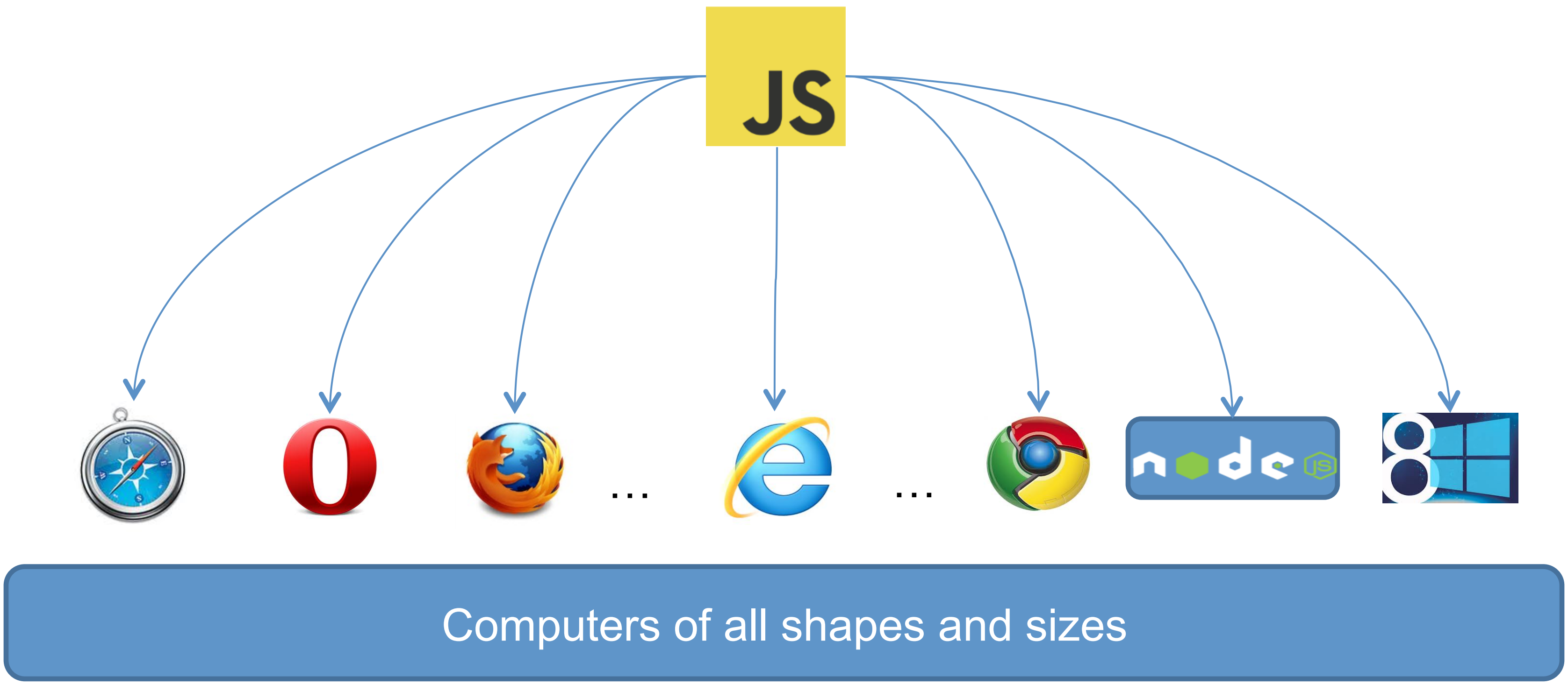
Taming JavaScript with F*

Nikhil Swamy

Microsoft Research

Everywhere!



But …

```
“hello” + 17 ~> “hello17”

‘0’ == false ~> true

{f:0}.g ~> undefined

' \t\r\n ' == 0 ~> true
```

…

huh?!

```
function foo(x) { return x + 1; }
function bar(x) { assert (foo(0)==1); }
```

Does this assertion always succeed?

```
foo = function (x) { return x; }
```

Not if you can change `foo`

REASONING ABOUT JAVASCRIPT IS HARD!

Does wrap intercept all sends with a whitelist check?

```
function wrap (rawSend){
    var whitelist =
        ['http://www.microsoft.com/mail',
         'http://www.microsoft.com/owa'] ;

    return function (target, msg) {
        if (whitelist[target])
          rawSend (target, msg);
       else throw("Rejected");
    }
}
```

```
Object.prototype[“evil.com”] =
true
```

If the context adds a field to Object.prototype, it can circumvent the `whiteList` check in `wrap(rawSend)`

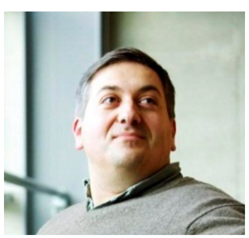
INSECURE!

Enter F* …

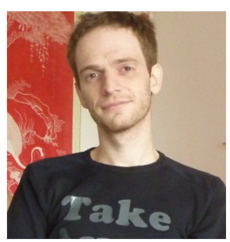
http://research.microsoft.com/fstar

An ML-like programming language with an SMT-based type system for program verification

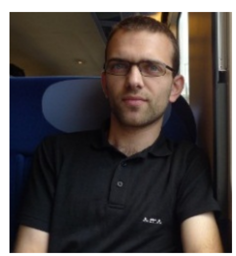
Juan Chen



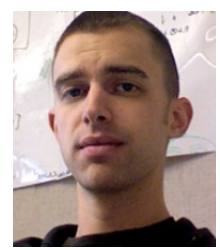
Cédric Fournet



Pierre-Yves Strub



Pierre Dagand

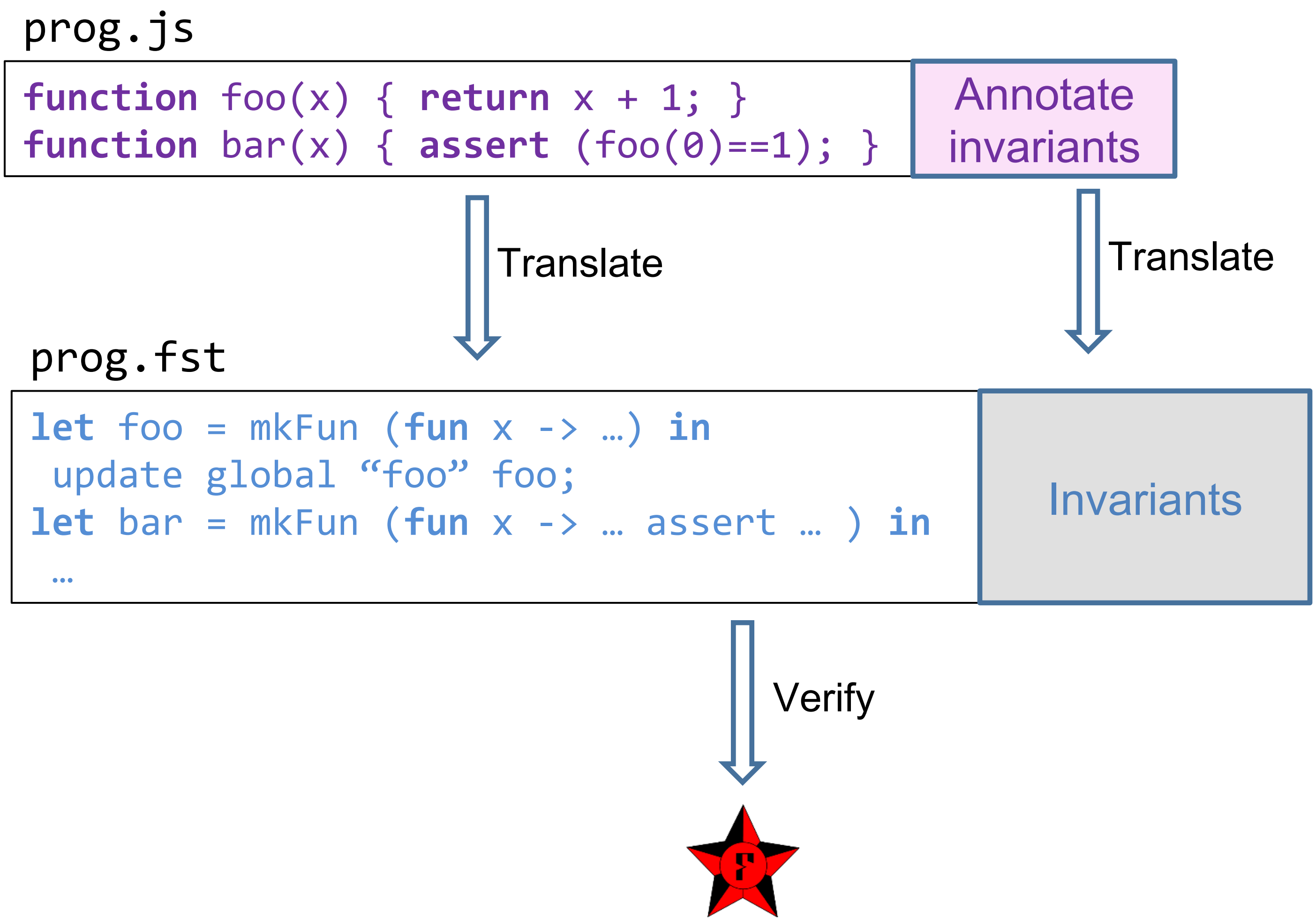


Cole Schlesinger

Joel Weinberger

Ben Livshits

1. Static assertion checking of JavaScript source



Verifying higher-order programs with the Dijkstra monad; PLDI 2013

2. A fully abstract compiler from F* to JavaScript

prog.fst

```
let foo x = x + 1
let bar = assert (foo 0 = 1)
```

⇓ Compile

prog.js

```
function () {
  function foo(x) { return x + 1; };
  assert (foo(0)==1);
} ()
```

Guaranteed to satisfy
all properties of prog.fst,
by construction

Fully abstract compilation to JavaScript; POPL 2013

If you must program in JavaScript:

F* can statically analyze your code for safety

Others tools can help too:
DJS by Chugh et al;
Gatekeeper by Livshits et al.; …

Otherwise, program in a nicer source language and compile safely to JS

JavaScript Semantics?

JS*: A semantics of JavaScript within F*

a.js

```
(x={f:'hello'},
 x.f)
```

JS2JS* ⇒

a.js.fst

```
open JSStar
update global (Str "x")
          (objlit [("f", Str "hello")]);
select (select global (Str "x")) (Str "f")
```

JSStar.fst

```
module JSStar
type dyn = Str : … | …
val select: o:dyn -> f:dyn -> …
val update: o:dyn -> f:dyn -> v:dyn -> …
val objlit: list (string * dyn) -> …
```

JS*: An instance of F*, where all free variables are defined by the module JSStar

JS2JS* based on LambdaJS; Guha et al.; ECOOP 2010

- The translation of every sub-term has type dyn in JS*
- Named functions are stored in the enclosing object
- Function calls involve lookups in a dynamically typed higher-order store
- Store is subject to arbitrary updates

JS

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

⇩ JS2JS*

JS*

```
> let foo = fun this args ->
            let x = select args (Str “0”) in
            update this (Str “g”)
                       (plus (select x (Str “f”)) (Num 1.0)) in
  update glob (Str “foo”) (Fun foo);

> let args = objLit [(Str “0”, objLit [(Str “f”, Num 0.0)])] in
  apply (select glob (Str “foo”)) glob args;

> update glob (Str “foo”) (Num 17.0)
```

```
module JSStar

type dyn =
   | Num : float -> dyn
   | Str : string -> dyn
   | Obj : ref (map string dyn) -> dyn
   …
   | Fun : (dyn -> dyn -> dyn) -> dyn
```

```
module JSStar
```

Refinement formula recovers static type information

//Refined type dyn

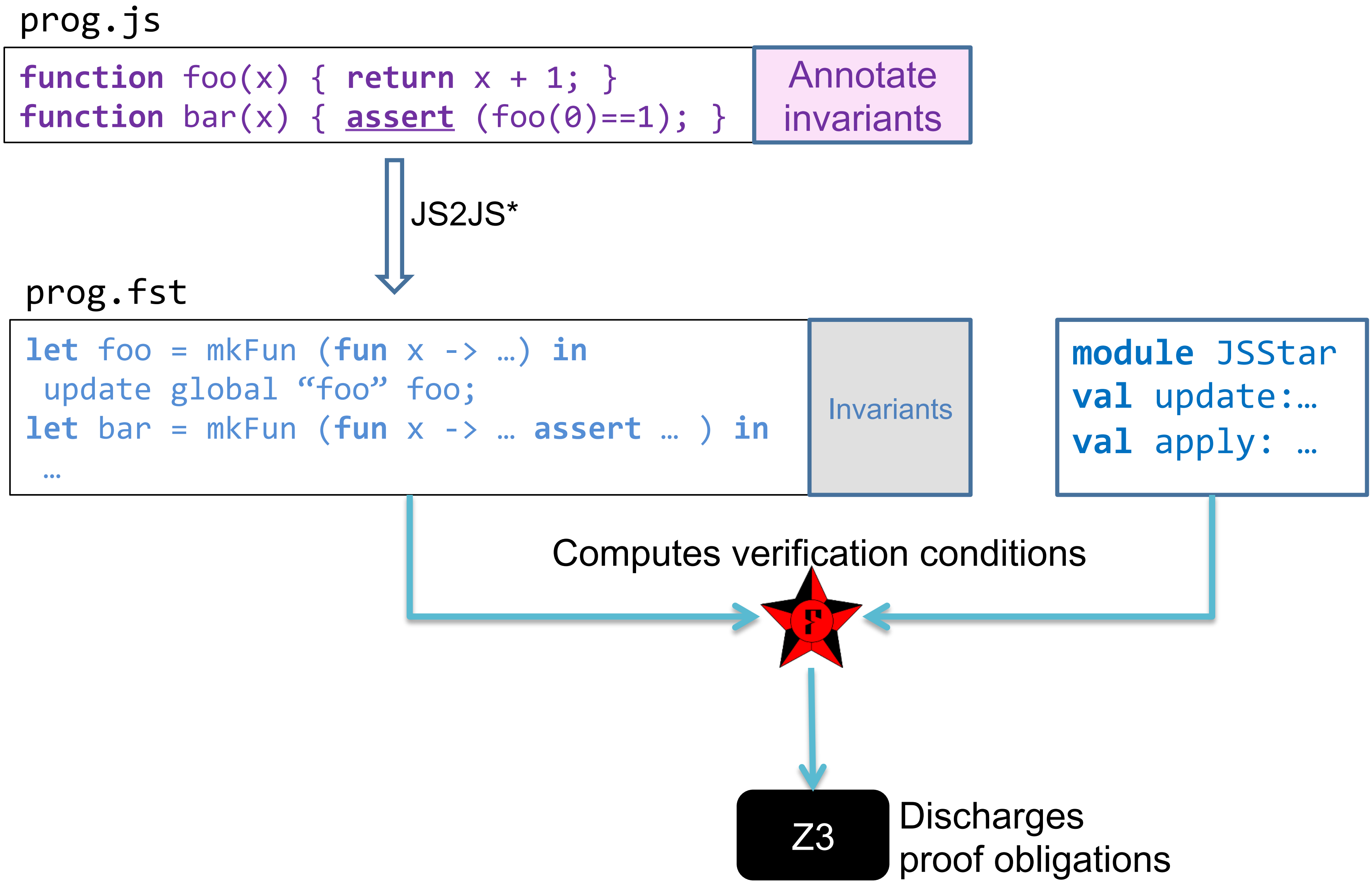
```
type dyn =
  | Num : float -> d:dyn{TypeOf d = float}
  | Str : string -> d:dyn{TypeOf d = string}
  | Obj : ref (map string dyn) -> d:dyn{TypeOf d = object}
  …
  | Fun : (x:dyn -> y:dyn -> ST dyn (WP x y))
        -> d:dyn{TypeOf d = Function WP}
```

Refinement for a function records the *weakest pre-condition* of the function

Dijkstra State Monad

The type of a computation that diverges or produces a `dyn` result satisfying `Post` when run in a heap satisfying `WP x y Post`

```
module JSStar

//Refined type dyn
type dyn =
   | Num : float -> d:dyn{TypeOf d = float}
   | Str : string -> d:dyn{TypeOf d = string}
   | Obj : ref (map string dyn) -> d:dyn{TypeOf d = object}
   …
   | Fun : (x:dyn -> y:dyn -> ST dyn (WP x y))
         -> d:dyn{TypeOf d = Function WP}

val select: o:dyn -> f:dyn -> ST dyn
    (Requires (\h. TypeOf o = object /\ HasField h o f))
    (Ensures (\res h0 h1. h0=h1 /\ res=SelectField h0 o f))
let select o f = //traverse prototype chains etc.

val update: o:dyn -> f:dyn -> v:dyn -> ST dyn
    (Requires (\h. TypeOf o = object))
    (Ensures (\res h0 h1. h1=UpdateField h0 o f b /\ res=Undef)
let update o f v = //traverse prototype chains etc.

…
```



Verifying higher-order programs with the Dijkstra monad; PLDI 2013

Name	LOC(JS)	TC (sec)	Description
JSStar	1,131	63.5	Runtime support for JavaScript
Untiny	59(9)	11.0	Send selected URL
Delicious	65(13)	11.3	Bookmark selected text
Passward	111(29)	42.7	Store and retrieve passwords
HoverMagn	60(23)	38.1	Magnify text under the cursor
Typograf	106(28)	65.5	Format text a user inputs
Facepalm	270(82)	718.0	Find contacts from Facebook

Verified for functional correctness

Verified for absence of runtime errors

Ok, but what about the context?

```
function foo(x) { return x + 1; }
function bar(x) { assert (foo(0)==1); }
  Requires \h. TypeOf h[glob[“foo”]]=Function WP …
  Ensures …
```

Writing JS functions that have trivial pre-conditions is really hard!

Pre-condition to ensure the correctness of wrap?

```
function wrap (rawSend){
    var whitelist =
        ['http://www.microsoft.com/mail',
         'http://www.microsoft.com/owa'] ;

    return function (target, msg) {
        if (whitelist[target])
          rawSend (target, msg);
       else throw("Rejected");
    }
}
```

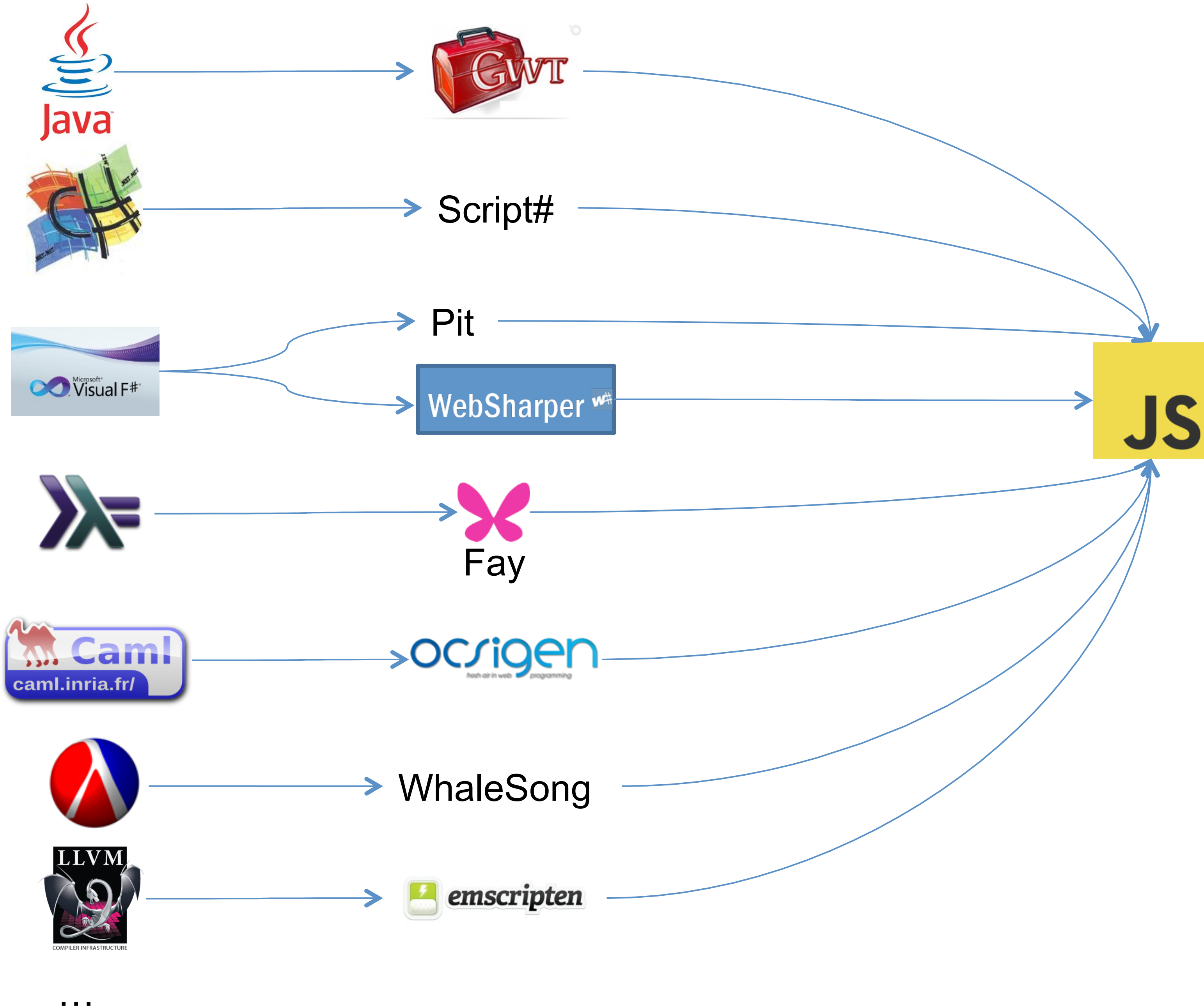
```
Object.prototype[“evil.com”] = true
```

What if you could instead write `wrap` in ML?
(Or your favorite high-level language)

```
//Obviously correct!
let wrap rawSend =
  let whitelist = ["http://talk.google.com";
                   "http://talk.live.com"] in
  let newSend target msg =
      if List.contains whitelist target
      then rawSend target msg
      else () in
  newSend
```

And we compiled it to JavaScript for you

That seems to be the way things are going …



```
module WebPage =
struct
  type t = …
  let doStuff …
end
```

WebPage.ml

ocsigen

```
<html>
<head>
<script src="google.com/jquery.js">
</script>

<script type="text/javascript">
  WebPage = {
     doStuff : function () {
      …
     }
  }
</script>

<script src="adz.com/clicktrack.js">
</script>
</head>
<body> … </body>
</html>
```

WebPage.html

Interactions between compiled ML code and JS can be problematic

```
let wrap rawSend =
  let whiteList =
        ["http://www.microsoft.com/mail";
         "http://www.microsoft.com/owa"] in

  fun target msg ->
   if mem target whiteList
   then rawSend target msg
   else failwith "Rejected"
```

Compile naively

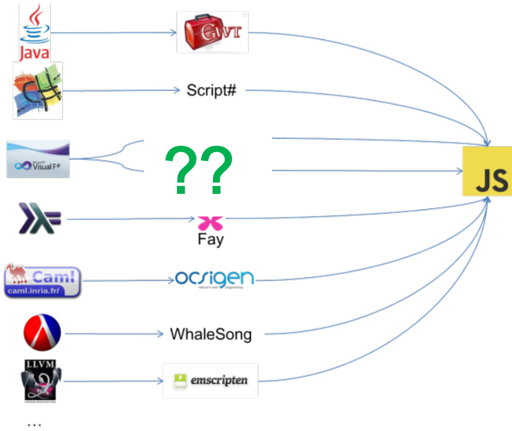
```
function wrap (rawSend){
    var whitelist =
        ['http://www.microsoft.com/mail',
         'http://www.microsoft.com/owa'] ;

    return function (target, msg) {
        if (whitelist[target])
          rawSend (target, msg);
       else throw("Rejected");
    }
}
```

```
Object.prototype["evil.com"] = true
```

If the context adds a field to Object.prototype, it can circumvent the `whiteList` check in `wrap(rawSend)`

INSECURE!

Need a compiler that:



Allows a programmer to reason in source semantics

Emits JavaScript that behaves like the source program

In ALL JavaScript contexts

Intuitively, this should be hard because JavaScript contexts can do more than (say) ML contexts

- Prototype poisoning
- getters/setters
- Implicit conversions
- Functions as objects
- An unusual calling convention
- `Function.toString`
- `arguments.callee.caller`
- `(new Error()).stack`
- `with`
- `eval`
- …

A fully abstract compiler from F* to JavaScript

Full abstraction is the ideal property for a translation:

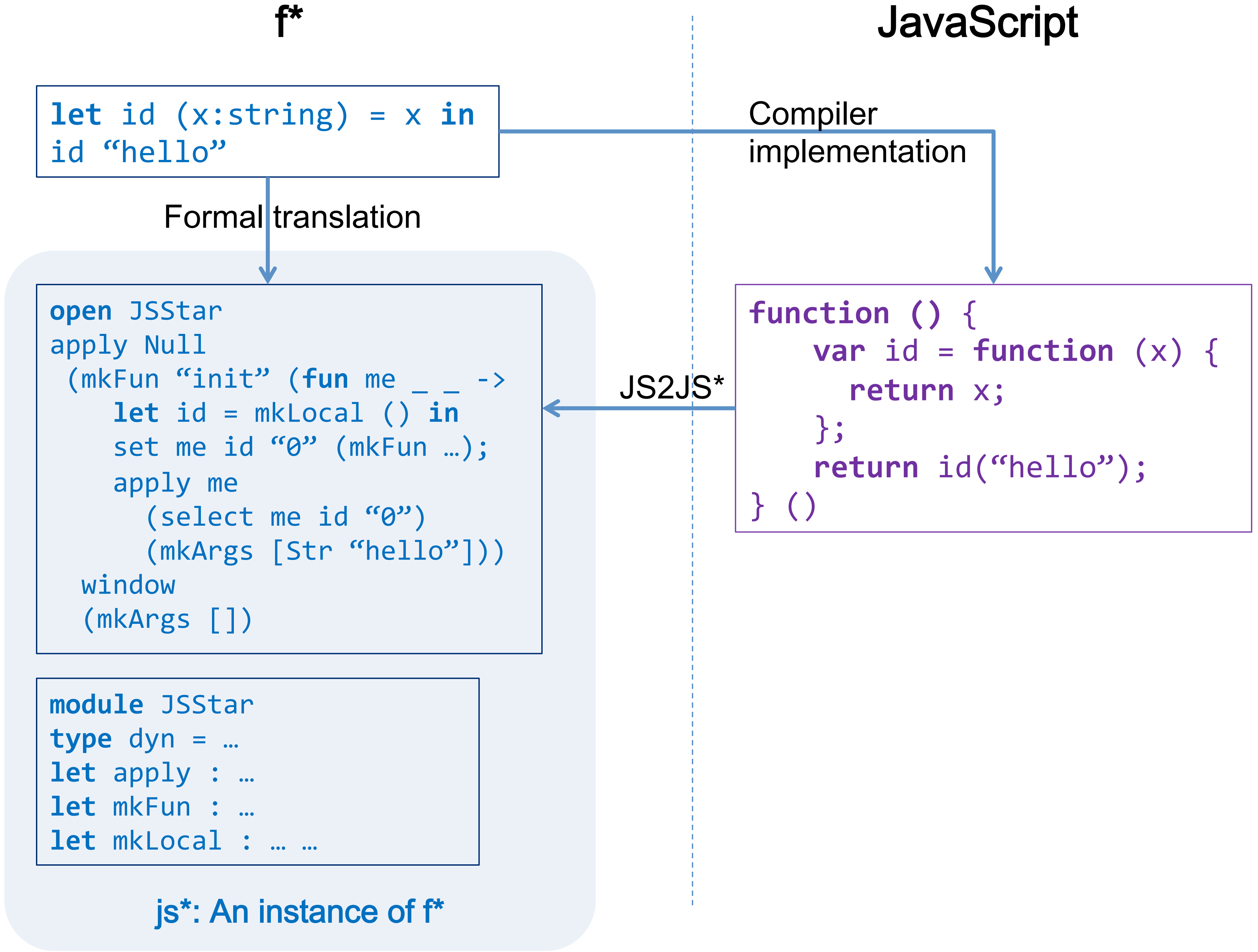
$$8e_1; e_2 : e_1 ¼_{f¤} e_2 \; (\;) \quad \text{compile}(e_1) ¼_{JS} \text{compile}(e_2)$$

Where for a language L :

e_1 ¼$_L$ e_2 ,

for all L-contexts E ;

outcome(E [e_1]) = outcome(E [e_2])



f*
JavaScript
let id (x:string) = x in
id “hello”
Compiler
implementation
Formal translation
open JSStar
apply Null
(mkFun “init” (fun me _ _ ->
let id = mkLocal () in
set me id “0” (mkFun …);
apply me
(select me id “0”)
(mkArgs [Str “hello”]))
window
(mkArgs [])
JS2JS*
function () {
var id = function (x) {
return x;
};
return id(“hello”);
} ()
module JSStar
type dyn = …
let apply : …
let mkFun : …
let mkLocal : … …
js*: An instance of f*

Source f*:

```
let id (x:string) = x in
id "hello"
```

¼ f¤

```
"hello"
```

Formal translation

Formal translation

Target js*:

```
open JSStar
apply Null
 (mkFun "init" (fun …)
 window
 (mkArgs []))
```

¼ js¤

```
open JSStar
Str "hello"
```

```
module JSStar
type dyn = Null | Str of string | …
let apply : …
let mkFun : …
let mkLocal : … …
```

Proof technique leveraging f*

(a) **Type-based verification of key invariants**

Heap shape invariant
Heap separation invariant
Type preservation lemma
Weak forward simulation

```
module JSStar

type dyn =
  | Num : float -> d:dyn{TypeOf d = float}
  | Str : string -> d:dyn{TypeOf d = string}
  | Obj : ref (map string dyn) -> d:dyn{TypeOf d = object}
  …
  | Fun : (x:dyn -> y:dyn -> ST dyn (WP x y))
         -> d:dyn{TypeOf d = Function WP}

val select: o:dyn -> f:dyn -> ST dyn
    (Requires (\h. TypeOf o = object /\ HasField h o f))
    (Ensures (\res h0 h1. h0=h1 /\ res=SelectField h0 o f))

val update: o:dyn -> f:dyn -> v:dyn -> ST dyn
    (Requires (\h. TypeOf o = object))
    (Ensures (\res h0 h1. h1=UpdateField h0 o f b /\ res=Undef)
…
```

Proof technique leveraging f*

(b) A new applicative bisimulation for contextual equivalence

- Supports higher-order functions, mutable state, exceptions, divergence, fatal errors
  - Same bisimulation applies to both source and target, since both are just instances of f*
- **Theorem**: Applicative bisimilarity (the largest bisimulation) is sound and complete w.r.t contextual equivalence
- **Theorem** (equivalence preservation): Formal translation of a bisimilar source f* configurations produces bisimilar js* configurations.
- **Theorem** (equivalence reflection): If the formal translation of any pair of source f* configurations produces bisimilar js* configurations, then source f* configurations are bisimilar.

Structure of the translation

Two phases

1. A compositional “light translation”
2. Type-directed defensive wrapping

The light translation:

$$[\![x]\!] \mapsto \texttt{x} \qquad [\![C_{\bar{t}\to T}\ \bar{v}]\!] \mapsto \{\texttt{"tag":"C",}\overline{\texttt{"i":}[\![v_i]\!]}\}$$

$$[\![e_1\ v_2]\!] \mapsto [\![e_1]\!]([\![v_2]\!]) \qquad [\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] \mapsto \texttt{(x=}[\![e_1]\!],[\![e_2]\!])$$

$$[\![\mathsf{ref}\ v]\!] \mapsto \{\texttt{"ref":}[\![v]\!]\} \qquad [\![v_1 := v_2]\!] \mapsto [\![v_1]\!]\texttt{.ref=}[\![v_2]\!]$$

$$[\![!v]\!] \mapsto [\![v]\!]\texttt{.ref} \qquad [\![\textbf{error}]\!] \mapsto \texttt{alert("error")}$$

$$[\![\lambda x{:}t.e]\!] \mapsto \texttt{function(x)}\{\overline{\texttt{var x=}\{\texttt{"0":0}\}}\texttt{;return}[\![e]\!]\texttt{;}\}\ \text{for}\ \bar{x} = \text{locals}(e)$$

$$[\![\textbf{match}\ v\ \textbf{with}\ C_{\bar{t}\to T}\ \bar{x} \to e_1\ \textbf{else}\ e_2]\!] \mapsto$$
$$\texttt{(}[\![v]\!]\texttt{.tag==="C")?}(\overline{\texttt{x\_i=}[\![v]\!]\texttt{["i"]}},[\![e_1]\!])\texttt{:}[\![e_2]\!]$$

Light translation

```
let id (x:string) = x in
id “hello”
```

```
let id = ref Undefined in
let f = fun me this args ->
   lookup !args “0” in
id := mkFun f;
apply !id window
   (mkArgs [(“0”, “hello”)])
```

Semantically complex!

- Heavy use of mutation
- Higher-order store
- Functions are objects too
- Unusual calling convention
- …

```
var id = function (x) {
    return x;
}
id(“hello”)
```

Syntactically simple

Interacting with a JS context

```
fun (b:bool) -> b
```

Light → translation

¼ f¤

```
fun b ->
 if b then true
 else false
```

Light → translation

Phase 2: Type-directed defensive wrappers:

```
↓t : t  -> Un
↑t : Un -> t
```

```
Compile(e:t) = ↓t ([[e]]) : Un
```

Un : “unconfidential/untrusted” values that can be disclosed to the context or provided by the context

Defensive wrappers: ↑t : Un -> t
↓t : t -> Un

Enforcing a strict heap separation

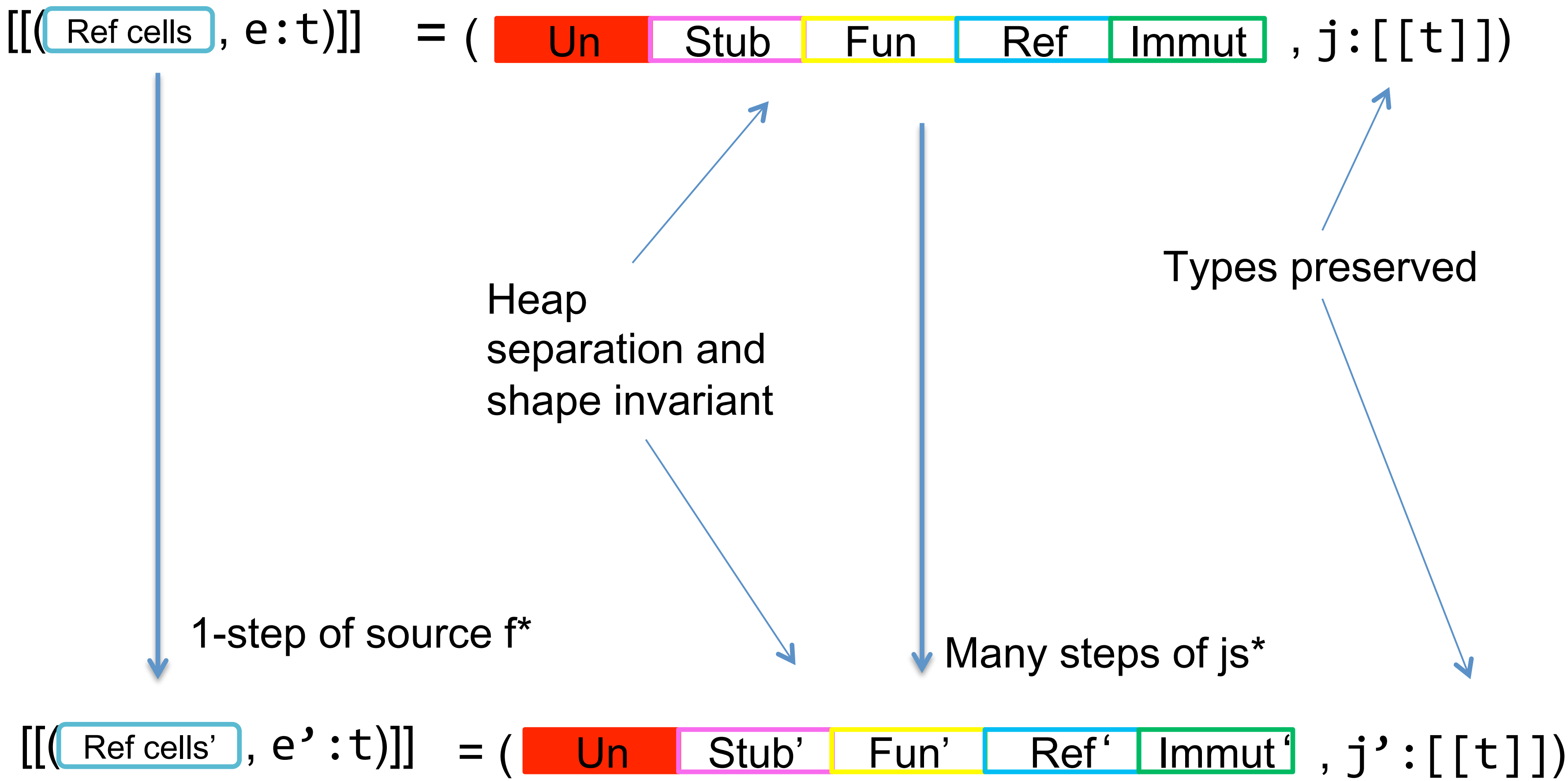
Down wrappers export values to the context:

```
↓bool      = function(b){return b;}
↓(a * b)   = function(p){return {0:↓a(p[“0”]), 1:↓b(p[“1”]);}
↓(a -> b)  = function(f){return function(x) {return ↓b(f(↑a(x)));};}
```

Up wrappers import values from the context:

```
↑bool      = function(b){return b?true:false;}
↑(a * b)   = function(p){return {0: ↑a(p[“0”]), 1:↑b(p[“1”]);}
↑(a -> b)  = …
```

Some properties of the translation
expressed using the types of F*
partially mechanized using the F* typechecker



Wrappers secure interactions with the context

```
fun (b:bool) -> b
```

¼ f¤

```
fun b ->
 if b then true
 else false
```

JS callbacks can walk the stack

```
(fun b f -> f() && b)
true
```

¼ f¤

Light translation →

```
function(b) { return
  function(f) {
        return (f() && b);};
}(true);
```

~~¼ js¤~~

```
fun f -> f()
```

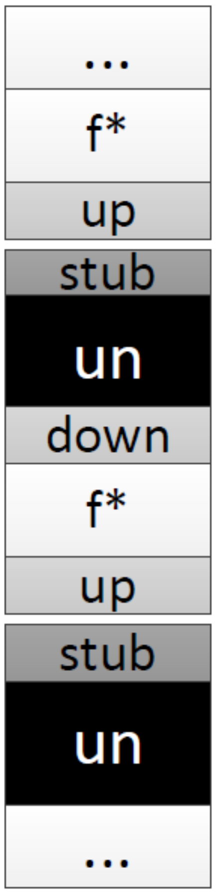
Light translation →

```
function(f) {return f();}
```

```
function(g) {
  return g(function() {
              arguments.callee.caller.arguments[0] = false;
              return true;});
}
```

Callback "stubs" prevent stack walks

↑(a -> b) =




```
function (f) {
  return function (x) {
    var z = ↓a(x);
    var y = undefined;
    function stub(b) {
      if (b) { stub(false); }
      else { y = ↑b(f(z)); }
    }
    stub(true);
    return y;
  };
}
```

See online paper for details

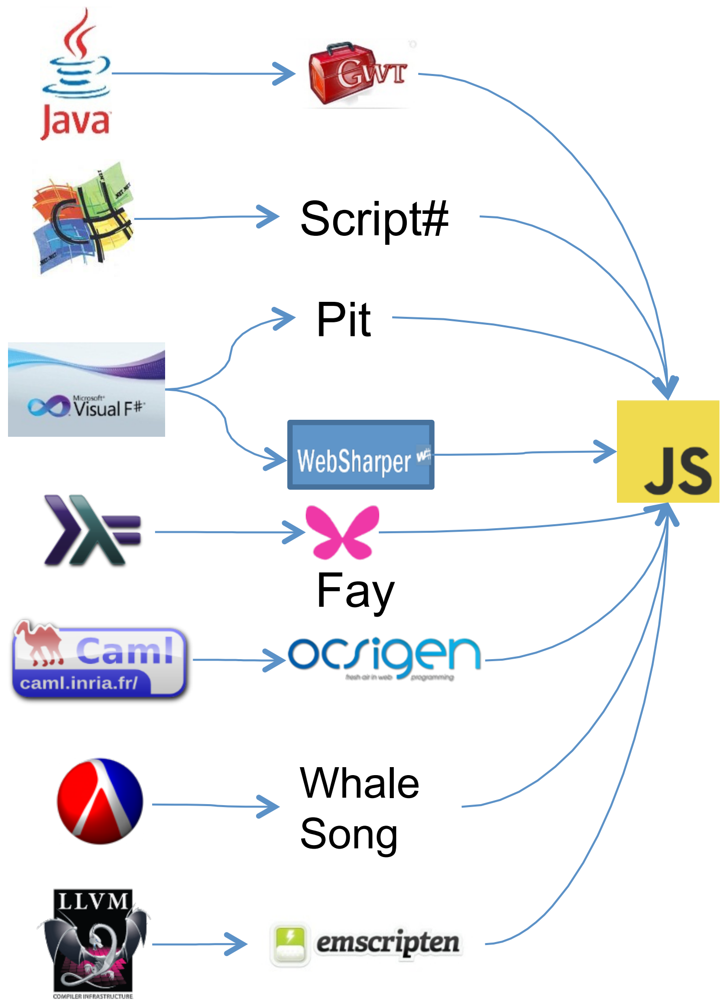
Limitations

- Side channels
  - Full abstraction can be broken with resource exhaustion/timing attacks
- Cannot import polymorphic functions
- Temporary limitations
  - Context can use exceptions
    - But not yet source programs
  - Implementation supports polymorphism
    - But not in our formalism yet

Demo

http://rise4fun.com/tutorials/FStar/jsStar

JS as a compiler target



- Use our wrappers for heap separation
- Allow your programmers to think in source semantics!

JS dialects / sub-languages

TypeScript

Static (gradual) typing
Better IDE support
Verification more feasible

```
class Greeter {
    who: string;
    constructor (w:string) {
        this.who=w;
    }
    greet() {
      return "Thank you, "
           +this.who;
    }
}
new Greeter("HCSS!").greet();
```

We provide solutions for programmers who:

… must write JavaScript code directly

… can generate JS from a language with a cleaner semantics

JavaScript: Not so bad after all … at least as a compilation target

http://research.microsoft.com/fstar

Experiments

- A mini-ML compiler bootstrapped in JavaScript
  - ~1.5 seconds to bootstrap in F#/.NET
  - ~3 seconds to bootstrap in IE 10
- Several small security examples
  - Secure local storage, API monitoring, etc.
- Online demo and “full-abstraction game”: http://rise4fun.com/tutorials/FStar/jsStar

Anyone for <script type=“text/ml”>?

THEOREM 1 (Full abstraction). *For all* f* *translations* $\vdash (v_0, v_1) : (t * t) \rightsquigarrow (e_0, e_1)$*, we have* $v_0 \approx v_1$ *if and only if* $\mathit{JSVerify}[\downarrow t\ e_0] \approx \mathit{JSVerify}[\downarrow t\ e_1]$.