JSkel - A JavaScript Formalization in Skeletal Semantics

Seminaire au vert

Adam Khayam
Advisors: Alan Schmitt, Tamara Rezk

Inria Bretagne Atlantique - Team Celtique

27/11/2020
The Question

Is it possible to make a “good” formalization of JavaScript specification?
The “good” formalization of a language

C’est quoi?

- Visually close to its specification;
- Executable;
- Maintainable;
- Prove properties of programs;
- Prove properties of the language;
- Reusable.
Why JavaScript?

JavaScript is everywhere

- Widely used for laptop and mobile web applications

https://madnight.github.io/github/#!/pull_requests/2020/1
Why JavaScript?

JavaScript matters for web security

- Executed within the Browser context;
- Interaction between different JS components;
  - Contents getting more and more interesting;
  - Complex applications have flaws;
    - XSS - Cross-Site Scripting,
    - CSRF - Cross-Site Request Forgery,
    - Server-Side JavaScript Injection,
    - Browser Extensions [1],
    - ...

A.Khayam
Inria Bretagne Atlantique

JSkel - A JavaScript Formalization in Skeletal Semantics
Why JavaScript?

JavaScript matters for web security

- Executed within the Browser context;
- Interaction between different JS components;
  + Contents getting more and more interesting;
  - Complex applications have flaws;
    - XSS - Cross-Site Scripting,
    - CSRF - Cross-Site Request Forgery,
    - Server-Side JavaScript Injection

Chrome 0-day exploit CVE-2019-13720 used in Operation WizardOpium [2]
Why JavaScript?

JavaScript is complex

- Multi-Paradigm;
- Objects as:
  - Literals;
  - Object Factories;
  - Functions;
- Prototype-based:
  - Complex behaviours;
  - Shared behaviours.
- Resiliency to errors, *i.e.*
  - complex syntax rules + type conversion semantics
  - redefinition of global methods
Why JavaScript?

JavaScript is complex

- Multi-Paradigm;
- Objects as:
  - Literals;
  - Object Factories;
  - Functions;
- Prototype-based;
  - Complex behaviours;
  - Shared behaviours.
- Resiliency to errors, i.e.
  - complex syntax rules + type conversion semantics
  - redefinition of global methods

```javascript
var x = (![]+[])[+!+[]]
    + ([] [[]+[])[+!+[]]+[]]
    + (+[[]]+[])[+!+[]]
    + (Number+""")[+[+!+[]]+[+!+[]]]
console.log(x)
//Result => 'adam'
```
Why JavaScript?

JavaScript is complex

- Multi-Paradigm;
- Objects as:
  - Literals;
  - Object Factories;
  - Functions;
- Prototype-based;
  - Complex behaviours;
  - Shared behaviours.
- Resiliency to errors, *i.e.*
  - complex syntax rules + type conversion semantics
  - redefinition of global methods

```javascript
var x = (![+[]][+!+[]] + ([[]]+[])[+!+[]]+[] + (![+[]][+!+[]] + (Number+""))[[+![]]+[+![]]])

console.log(x)
//Result => 'adam'
```

```javascript
var o = {}
o.toString = function () {
o.toString = function () {
  return "Evil muahah 3:-)"
} return "Good Guy O:-)"

console.log("Let's test : "+ o)
console.log("Let's use the Good guy : "+ o)
// Results:
// Let's test : Good Guy O:-)  
// Let's use the Good guy : Evil muahah 3:-) 
```
JavaScript is complex

```javascript
var o = {};

o.toString = function () {
    o.toString = function () {
        return "Evil muahah 3:-)"
    }
    return "Good Guy 0:-)"
};

console.log("Let's test : " + o);
console.log("Let's use the Good guy : " + o);

// Results:
// Let's test : Good Guy 0:-)
// Let's use the Good guy : Evil muahah 3:-)
```
JavaScript is complex

```javascript
var x = (![]+[])[++!+[]]+ (![]+[])[++!+[]]+ (Number+"")[++!+[]]+[+!+[]][+!+[]]

console.log(x)
//Result => 
```

```javascript
var o = {}
o.toString = function() {
    o.toString = function() {
        return "Evil muahah 3:-)"
    }
    return "Good Guy O:-)"
}

console.log("Let's test : " + o)
console.log("Let's use the Good guy : " + o)
// Results: 
// Let's test : Good Guy O:-) 
// Let's use the Good guy : Evil muahah 3:-)
```

A.Khayam

Inria Bretagne Atlantique

JSkel - A JavaScript Formalization in Skeletal Semantics
 ECMAScript is the specification of JavaScript

- Verbose;
  - 28 chapters;
  - 6 appendices.

- Vernacular pseudo-code fashion;

- Loosely typed;

- Parts of the specification are left as implementation choices;

- “Every story has a beginning, a middle, and an end. Not necessarily in that order.” - Tim Burton
"We choose to go to the Moon... We choose to go to the Moon in this decade and do the other things, not because they are easy, but because they are hard; ..."
Why JavaScript?

But..

This hasn’t been already done by someone else?
This hasn’t been already done by someone else?

YES, but...
JavaScript-Related Works

- **KJS** [4]

- A K [3] formalization of ECMAScript 5.1;
- Executable semantics;
- Framework suitable to analyse programs;
- Passes all the ecma262 test suite;
- Used for finding bugs.

- Does it look like the ECMAScript?
  + Is it executable?
  - Is it maintainable?
  + Allow to reason on programs?
  - Allow to prove language properties?
  - Is it still used?
JavaScript-Related Works

JSCert [7]

- A Pretty-Big-Step [5] formalization of ECMAScript 5.1;
- Correctness between the inductive and the recursive definition of ES;
- Extraction of an OCaml interpreter;
  - Trust.
- Representation of the core language;
  - Good results in the ecma262 test.
- JSExplain [6];
- Used for finding bugs.

Does it look like the ECMAScript?
+ Is it executable?
~ Is it maintainable?
+ Allow to reason on programs?
~ Allow to prove language properties?
~ Is it still used?
JavaScript-Related Works

\( \lambda_{JS} \) [8]

- A desugared representation of ECMAScript 3 and 5;
- Embodies JavaScript essentials;
- Feature-based representation;
- Good results in the ecma262 test;
- Does it look like the ECMAScript?
  - Is it executable?
  - Is it maintainable?
  + Allow to reason on programs?
  + Allow to prove language properties?
  - Is it still used?
The big common "-" of $K_{JS}$, JSCert, and $\lambda_{JS}$

Trying to address all the "good" formalization requirements with one single answer!
The big common "-" of $K_{JS}$, JSCert, and $\lambda_{JS}$

Trying to address all the “good” formalization requirements with one single answer!

Possible solution: Decouple the semantics from its use
1 Introduction
   The "good" formalization of a language
   Why JavaScript?
   JavaScript-Related Works

2 Dead bone tell no tales
   The Semantics
   Making Skeletons Alive - Natural Interpretation

3 JSkel
   Running example
   State monad
   ECMAScript Error-Handling monad
   Control-Flow monad
   The ECMAScript monad
   The new GetValue(V) - ECMAScript 2021 - November 2020 update
   The point

4 Conclusions
   Contribution summary
   Future

5 Bibliography
The Semantics

Skeletal[9]

- Framework for capturing the structure of semantics;
- Familiar syntax;
- Easier to represent algorithmic and rule-based semantics;
- Generic definition of interpretations;
- Proof techniques to relate interpretations
- Coq generation
The Semantics

**while Big-Step Rule**

\[
\begin{align*}
\sigma, e \downarrow b & \quad b = \text{true} & \sigma, t \downarrow \sigma'' & \quad \sigma'', \text{while } e \ t \downarrow \sigma' \\
\sigma, \text{while } e \ t \downarrow \sigma' & \quad \sigma, e \downarrow b & b = \text{false} & \sigma, \text{while } e \ t \downarrow \sigma
\end{align*}
\]

**while in skel**

```javascript
term eval_stmt : (state, stmt) → state =
    λ (σ, t) : (state, stmt) →
        branch let ... = t in
            ...
        or let While (e, t) = t in
            let b = eval_expr (σ, e) in
            branch isTrue b;
                let σ'' = eval_stmt (σ, t) in
                eval_stmt (σ'', While (e, t))
            or isFalse b;
                σ
            end
        end
```
The skel language

**TERM** \( t \) ::= \( x_i \mid C \ t \mid (t, \ldots, t) \mid \lambda x : \tau \cdot S \)

**SKELETON** \( S \) ::= \( x_i \ t_1 \ldots t_n \mid \text{let } p = S \text{ in } S \mid (S..S) \mid t \)

**PATTERN** \( p \) ::= \( x_i \mid _- \mid C \ p \mid (p, \ldots, p) \)

**TYPE** \( \tau \) ::= \( b \mid \tau \rightarrow \tau \mid (\tau, \ldots, \tau) \)
necroml [10]  -If you can't get rid of the skeleton in your closet, you'd best teach it to dance.-  G.B. Shaw

- Takes a skel file, and generates an OCaml functor
- Instantiation of unspecified types and terms is needed
- It is one of the generators tools.
  - necrotex, necrocoq, ...
necroml [10]  
-If you can’t get rid of the skeleton in your closet, you’d best teach it to dance.- G.B. Shaw

- Takes a skel file, and generates an OCaml functor
- Instantiation of unspecified types and terms is needed
- It is one of the generators tools.
  - necrotex, necrocoq, ...
1 Introduction
   The "good" formalization of a language
   Why JavaScript?
   JavaScript-Related Works

2 Dead bone tell no tales
   The Semantics
   Making Skeletons Alive - Natural Interpretation

3 JSkel
   Running example
   State monad
   ECMAScript Error-Handling monad
   Control-Flow monad
   The ECMAScript monad
   The new GetValue(V) - ECMAScript 2021 - November 2020 update
   The point

4 Conclusions
   Contribution summary
   Future

5 Bibliography
### ECMA's GetValue(V)

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
5. If IsPropertyReference(V) is true, then
   a. If HasPrimitiveBase(V) is true, then
      i. Assert: In this case, base will never be undefined or null.
      ii. Set base to ! ToObject(base).
   b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. Else,
   a. Assert: base is an Environment Record.
   b. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V)).
Running example

ECMA’s GetValue(V) with some annotations

1. **ReturnIfAbrupt(V).**
2. **If** Type(V) **is not Reference, return V.**
3. **Let** base be **GetBase(V).**
4. **If** IsUnresolvableReference(V) **is true, throw a ReferenceError exception.**
5. **If** IsPropertyReference(V) **is true, then**
   a. **If** HasPrimitiveBase(V) **is true, then**
      i. **Assert:** In this case, base will never be undefined or null.
      ii. **Set** base to !ToObject(base).
   b. **Return** ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. **Else,**
   a. **Assert:** base is an Environment Record.
   b. **Return** ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V)).
GetValue(V)’s Skeleton

term getValue : ( v : out<valref>) -> st<out<value>> =

let%cf_out result =
  let%returnIfAbrupt v = v in
  branch valref_Type(v, T_Reference);%f let Value v = v in ret<value> v
  or valref_Type(v, T_Reference);%t cont<value>()
  end;%bind

let Reference v = v in let base = getBase(v) in
branch isUnresolvableReference v;%t let%throw re = referenceError<value> in re
  or isUnresolvableReference v;%f cont<value>()
  end;%bind

branch isPropertyReference v;%t
  let%bind base =
    branch hasPrimitiveBase v;%t let R_Value base = base in
      _orb<(*types*)> val_Type (base, T_Undefined) val_Type (base, T_Null);%assF
      let%b base = toObject(base) in assign<loc_Object, value>(base)
    or hasPrimitiveBase v;%f
      let R_Value (Obj base) = base in assign<loc_Object, value>(base)
    end in

  let name = getReferencedName(v) in let%q thisVal = getValue(v) in
  let%q v' = o_Get(base, name, thisVal) in ret<value> v'
  or isPropertyReference v;%f let ref_Type(base, T_R_EnvironmentRecord);%assT let R_EnvironmentRecord base = base in
    let name = getReferencedName(v) in let strict = isStrictReference(v) in
    let%q v' = er_GetBindingValue(base, name, strict) in ret<value> v'
  end in

result

A.Khayam

Inria Bretagne Atlantique

JSkel - A JavaScript Formalization in Skeletal Semantics
State monad

- It is a Record;
- ES environment:
  - \( M_\alpha : \text{location} \rightarrow \alpha, \alpha \in \{\text{ExecutionContext}, \text{EnvironmentRecord}, \text{Object}, \text{ScriptRecord}, \text{ModuleRecord}, \text{RealmRecord}\}; \)
  - ExecutionContext Stack.
- It is a regular state monad;

\[
\begin{align*}
\text{type} & \quad \text{state} \quad (*\text{left unspecified}*) \\
\text{type} & \quad \text{st}<\alpha> := \text{state} \rightarrow (\alpha, \text{state}) \\
\text{term} & \quad \text{st\_bind}<\alpha,\beta>: (v: \text{st}<\alpha>) \rightarrow (f: \alpha \rightarrow \text{st}<\beta>) \rightarrow \text{st}<\beta> = \\
& \quad \lambda\ s:\ \text{state}\ \rightarrow\ \\
& \quad \quad \text{let}\ (v', s') = v\ s\ \text{in}\ \\
& \quad \quad f\ v'\ s' \\
\text{term} & \quad \text{st\_ret}<\alpha>: (v: \alpha) \rightarrow \text{st}<\alpha> = \\
& \quad \lambda\ s:\ \text{state}\ \rightarrow\ (v, s)
\end{align*}
\]
ECMAScript Error-Handling monad

ECMA's `GetValue(V)`

1. `ReturnIfAbrupt(V)`.
2. If `Type(V)` is not `Reference`, return `V`.
3. Let `base` be `GetBase(V)`.
4. If `IsUnresolvableReference(V)` is `true`, throw a `ReferenceError` exception.
5. If `IsPropertyReference(V)` is `true`, then
   a. If `HasPrimitiveBase(V)` is `true`, then
      i. **Assert**: In this case, `base` will never be `undefined` or `null`.
      ii. Set `base` to `ToObject(base)`.
   b. Return `base.[[Get]](GetReferencedName(V), GetThisValue(V))`.
6. Else,
   a. **Assert**: `base` is an `Environment Record`.
   b. Return `base.GetBindingValue(GetReferencedName(V), IsStrictReference(V))`.
ECMAScript Error-Handling monad

Described by ECMA | Proscribed by ECMA

- `returnIfAbrupt` | `Absent`
- `Assert` !

**Morale:**

- **Success** - "Okay, Houston ... we’ve had a problem"
- **Anomaly** - "Mayday, Mayday!"

```typescript
import { st } from './semantic-types';

// type definitions

type out<α> = Success completionRecord<α> | Anomaly anomaly;

// terms

term (?|!)<α,β> : st<out<α>> → (α → st<out<β>>)) → st<out<β>>
```

---

A.Khayam  

Inria Bretagne Atlantique

**JSkel - A JavaScript Formalization in Skeletal Semantics**
The Completion Record and the Error-Handling shorthands

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Type]]</td>
<td>One of normal, break, continue, return, or throw</td>
</tr>
<tr>
<td>[[Value]]</td>
<td>Any ECMAScript language value or empty</td>
</tr>
<tr>
<td>[[Target]]</td>
<td>Any ECMAScript string or empty</td>
</tr>
</tbody>
</table>

- It **should** contain only ES values or empty;
  - Polymorphic.
- Either **normal** or **abrupt**:
  - Break, Continue, Return, Throw;

```javascript
type completionType =
| Normal
| Break
| Continue
| Return
| Throw

type completionValue<α> =
| 0k α
| Abruption maybeEmpty<value>

type completionTarget :=
maybeEmpty<string>

type completionRecord<α> = (_Type_ : completionType,
_Value_ : completionValue<α>,
_Target_ : completionTarget)
```
ECMAScript Error-Handling monad

? and ! as monadic binders

?\langle\alpha,\beta\rangle v f

!\langle\alpha,\beta\rangle v f

let Anomaly e = v'

\vdash e : anomaly

retAnomaly\langle\beta\rangle(e)

let e = AbruptAnomaly (abruptionCoerce\langle\alpha,()\rangle v'')

\vdash e : anomaly

let \begin{align*}
\text{e} &= \text{abruptionCoerce}_{\langle\alpha,\beta\rangle} v'' \\
\text{st_retout}_{\langle\beta\rangle} (\text{Success e})
\end{align*}

\vdash e : \text{completionRecord}_{\langle\beta\rangle}

Normal?

\vdash v'' : \text{completionRecord}_{\langle\alpha\rangle}

\vdash v''' : \alpha

f v'''

\begin{aligned}
\text{term} & (\?|!\langle\alpha,\beta\rangle) : \text{st}\langle\text{out}\langle\alpha\rangle\rangle \rightarrow (\alpha \rightarrow \text{st}\langle\text{out}\langle\beta\rangle\rangle) \rightarrow \text{st}\langle\text{out}\langle\beta\rangle\rangle}
\end{aligned}
### ECMA's GetValue(V)

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
5. If IsPropertyReference(V) is true, then
   a. If HasPrimitiveBase(V) is true, then
      i. Assert: In this case, base will never be undefined or null.
      ii. Set base to ! ToObject(base).
   b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. Else,
   a. Assert: base is an Environment Record.
   b. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V))
Control-flow monad

(*Naive implementation*)

\[
\text{branch } \text{foo; } \%t \ 1 \ (*1*) \\
\text{or } \text{bar; } \%t \ 2 \ (*2*) \\
\text{or } \text{baz; } \%t \ 3 \ (*3*) \\
\text{or } \text{foo; } \%f \ \text{bar; } \%f \ \text{baz; } \%f \ 4 \ (*4*)
\]

(*Going in depth*)

\[
\text{branch } \text{foo; } \%t \ 1 \ (*1*) \\
\text{or } \text{foo; } \%f \\
\text{branch } \text{bar; } \%t \ 2 \ (*2*) \\
\text{or } \text{bar; } \%f \\
\text{branch } \text{baz; } \%t \ 3 \ (*3*) \\
\text{or } \text{baz; } \%f \ 4 \ (*4*) \\
\text{end}
\]

\text{end}

\[
\text{term } (t<\alpha>|f<\alpha>) : \text{boolean } \rightarrow (\text{boolean } \rightarrow \alpha) \rightarrow \alpha
\]
Control-Flow monad

\[
\text{type } \text{controlFlow}<\alpha, \beta> = \\
| \text{ContinueControl } \alpha \\
| \text{ReturnControl } \beta
\]

\[
\text{term } \text{cf_cont}<\beta> : () \to \text{controlFlow}<(), \beta> (* \text{green} *)
\]

\[
\text{term } \text{cf_ret}<\beta> : \beta \to \text{controlFlow}<(), \beta> (* \text{aluminium} *)
\]

\[
\text{term } \text{cf_assign}<\alpha, \beta> : \alpha \to \text{controlFlow}<\alpha, \beta> (* \text{orange} *)
\]

\[
\text{term } \text{cf_bind}<\alpha, \beta, \gamma> : \text{controlFlow}<\alpha, \beta> \to (\alpha \to \text{controlFlow}<\gamma, \beta>) \to \text{controlFlow}<\gamma, \beta>
\]

\[
\text{term } \text{cf_res}<\alpha, \beta, \gamma> : \text{controlFlow}<\alpha, \beta> \to (\alpha \to \beta) \to \beta
\]

```
let %cf_res result =
  branch foo; %t cf_ret<int> 1 (* 1 *)
  or foo; %f cf_cont<int> () end; %cf_bind

branch bar; %t cf_ret<int> 2 (* 2 *)
  or bar; %f cf_cont<int> () end; %cf_bind

branch bar; %t cf_ret<int> 3 (* 3 *)
  or bar; %f cf_cont<int> () end; %cf_bind

cf_ret<int> 4 (* 4 *)
```

in result
Control-Flow monad

```
type controlFlow<α,β> =  
  | ContinueControl α  
  | ReturnControl β

term cf_cont<β> : () → controlFlow<(),β> (*green*)

term cf_ret<β> : β → controlFlow<(),β> (*aluminium*)

term cf_assign<α,β> : α → controlFlow<α,β> (*orange*)

term cf_bind<α,β,γ> : controlFlow<α,β> → (α → controlFlow<γ,β>) → controlFlow<γ,β>

term cf_res<α,β,γ> : controlFlow<α,β> → (α → β) → β
```

If foo is true Return 1

Or foo;cf_cont<int> () end;cf_bind

branch foo;%t cf_ret<int> 1(*1*)
or    foo;%f cf_cont<int> () end;%cf_bind

branch bar;%t cf_ret<int> 2(*2*)
or    bar;%f cf_cont<int> () end;%cf_bind

branch bar;%t cf_ret<int> 3(*3*)
or    bar;%f cf_cont<int> () end;%cf_bind

in result

The or branch is verbose!
The ECMAScript monad

ECMA's GetValue(V)

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
5. If IsPropertyReference(V) is true, then
   a. If HasPrimitiveBase(V) is true, then
      i. Assert: In this case, base will never be undefined or null.
      ii. Set base to !ToObject(base).
   b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. Else,
   a. Assert: base is an Environment Record.
   b. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V))
What if we want to add error informations while we are dealing with the control-flow?

\[
\begin{align*}
\text{let} & \%\text{cf}\_\text{out} \ \text{result} = \\
& (*1*) \text{branch} \ \text{foo};& \%\text{p}\_\text{T} \ \text{let} \%\text{throw} \ fe=\text{fooError}<\text{int}> \ \text{in} \ fe \ \text{end};& \%\text{bind} \\
& (*2*) \text{branch} \ \text{bar};& \%\text{p}\_\text{T} \ \text{ret}<\text{int}> \ 2 \ \text{end};& \%\text{bind} \\
& (*3*) \text{branch} \ \text{bar};& \%\text{p}\_\text{T} \ \text{ret}<\text{int}> \ 3 \ \text{end};& \%\text{bind} \\
& (*4*) \ \text{ret}<\text{int}> \ 4 \\
\text{in} \ \text{result}
\end{align*}
\]

\[
\begin{align*}
\text{term} & \ \text{cf}\_\text{out}<\beta,\gamma> : \text{st}<\text{controlFlow}<\beta>,\beta>>> \rightarrow (\beta \rightarrow \gamma) \rightarrow \text{st}<\gamma> \\
\text{term} & \ \text{bind}<\alpha,\beta,\gamma> : \text{st}<\text{controlFlow}<\alpha,\text{out}<\beta>>> \rightarrow (\alpha \rightarrow \text{st}<\text{controlFlow}<\gamma, \text{out}<\beta>>>) \rightarrow \text{st}<\text{controlFlow}<\gamma, \text{out}<\beta>>> \\
\text{term} & \ \text{cont}<\beta> : () \rightarrow \text{st}<\text{controlFlow}<\beta>, \text{out}<\beta>>> (* \ b-y-g-a *) \\
\text{term} & \ \text{ret}<\beta> : \beta \rightarrow \text{st}<\text{controlFlow}<\beta>, \text{out}<\beta>>> (* \ b-y-g-a *) \\
\text{term} & \ \text{throw}<\beta> : (() \rightarrow \text{st}<\text{out}<\beta>>>) \rightarrow (\text{st}<\text{controlFlow}<\beta>, \text{out}<\beta>>> \rightarrow \text{st}<\text{controlFlow}<\beta>, \text{out}<\beta>>>) \rightarrow \\
\text{term} & \text{st}<\text{controlFlow}<\beta>, \text{out}<\beta>>> (* \ b-y-g-a *) \\
\text{term} & \ \text{assign}<\alpha,\beta> : \alpha \rightarrow \text{st}<\text{controlFlow}<\alpha,\text{out}<\beta>>> (* \ b-y-g-o *) \\
\text{term} & \text{p}\_\text{T}<\alpha>|\text{p}\_\text{F}<\alpha> : \text{boolean} \rightarrow (\text{boolean} \rightarrow \text{st}<\text{controlFlow}<\beta>, \text{out}<\alpha>>>) \rightarrow \text{st}<\text{controlFlow}<\beta>, \text{out}<\alpha>>> (*\text{partiality}*) \\
\text{term} & \text{fooError}<\alpha> : () \rightarrow \text{st}<\text{out}<\alpha>>>
\end{align*}
\]
1-to-1 comparison

With the introduction of records in skel, and partial branches in JSkel

```javascript
let cf_out result =
  let returnIfAbrupt v = v in
  branch valref_Type(v, T_Reference);%p_F
  let Value v = v in ret<value> v end;%bind
  let Reference v = v in
  isUnresolvableReference v;%p_T
  let throw re = referenceError<value> in re end;%bind
  branch isPropertyReference v;%t
    let R_Value v_base = v_base in
    let%b baseObj = toObject(v_base) in
    let%q thisVal = getThisValue(v) in
    let%q v' = o_Get(baseObj, v.referencedName, thisVal) in
    ret<value> v'
  or isPropertyReference v;%f
    let base be v.base in
    ref_Type(base, T_R_EnvironmentRecord);%assT
    let R_EnvironmentRecord base = base in
    let%q v = er_GetBindingValue(base, v.referencedName, v.strict) in
    ret<value> v'
end
in result
```

- `ReturnIfAbrupt(V)`.  
- If `V` is not `Reference Record`, return `V`.  
- If `IsUnresolvableReference(V)` is `true`, throw a `ReferenceError` exception.  
- If `IsPropertyReference(V)` is `true`, then  
  - Let `baseObj` be ! `ToObject(V.[[Base]])`  
  - Return ? `baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V))`.  
- Else,  
  - Let `base` be `V.[[Base]]`.  
  - Assert: `base` is an `Environment Record`.  
The point

Current state

- Semantics
  - ECMAScript Data Types and Values;
  - Abstract Operations
  - Executable Code and Execution Contexts;
  - Ordinary and Function Objects;
  - Implementation of ECMAScript Language sections.

- Interpreter
  - Runs basic programs;
    - Too many unspecified terms;
  + Parsing
    - FlowParser - Facebook;
      + Compliant with the SpiderMonkey API;
    - Transformation for getting an ECMAScript AST;
      - Weak assumption: A good transformation is the one resulting in a well-typed AST.
Is JSkel a “good” formalization?

+ Does it look like the ECMAScript?
+ Is it executable?
+ Is it maintainable?

∼ Allow to reason on programs? necrocoq

∼ Allow to prove language properties? necrocoq

+ It will be used? At least by us :p
The point

Is JSkel a “good” formalization?

+ Does it look like the ECMAScript?
+ Is it executable?
+ Is it maintainable?

~ Allow to reason on programs? necrocoq
~ Allow to prove language properties? necrocoq
+ It will be used? At least by us :p

Indeed..
1 Introduction
   The "good" formalization of a language
   Why JavaScript?
   JavaScript-Related Works

2 Dead bone tell no tales
   The Semantics
   Making Skeletons Alive - Natural Interpretation

3 JSkel
   Running example
   State monad
   ECMA Script Error-Handling monad
   Control-Flow monad
   The ECMAScript monad
   The new GetValue(V) - ECMAScript 2021 - November 2020 update
   The point

4 Conclusions
   Contribution summary
   Future

5 Bibliography
Conclusions

First year:

• Iteratively formalize JavaScript semantics;

• Defining a complex environment that simplifies writing JavaScript semantics;

• Design some of the necro features in order to simplify the ongoing formalization.

Publications:

Immediate future

What do we need for running tests?

1. Get up-to-date with ECMAScript 2021 November release;
   - A more precise definition of data structures;
   - A summary check shows that is slimmer;
   - We can specify more in the semantics, reducing implementation dependant choices;
   - skel has records now!
   - This work is really mechanic!

2. Complete the FlowParser transformations.
Future Work

• Produce an interpreter for JavaScript;

• Pave the way for analysis on JavaScript programs and its specification;

• How can we exploit the Coq formalization?
  • Proving that some invariants are preserved (i.e. Assertions);
  • Correctness proof between the concrete interpreter and the coq counterpart;
  • Analysis of small JavaScript programs.

• Produce a semantics for multi-tier languages;
  • Hop.js.
1 Introduction
   - The "good" formalization of a language
   - Why JavaScript?
   - JavaScript-Related Works

2 Dead bone tell no tales
   - The Semantics
   - Making Skeletons Alive - Natural Interpretation

3 JSkel
   - Running example
   - State monad
   - ECMAScript Error-Handling monad
   - Control-Flow monad
   - The ECMAScript monad
   - The new GetValue(V) - ECMAScript 2021 - November 2020 update
   - The point

4 Conclusions
   - Contribution summary
   - Future

5 Bibliography
References I


References II

