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

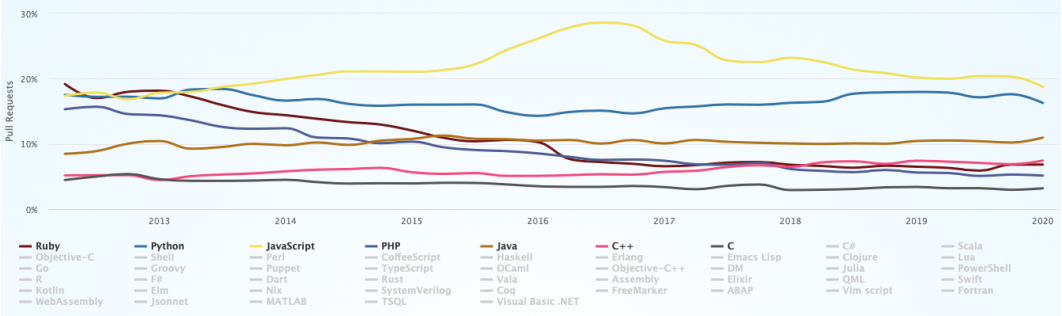
Is it possible to make a "good" formalization of JavaScript specification?

C'est quoi?

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

JavaScript is everywhere

- Widely used for laptop and mobile web applications



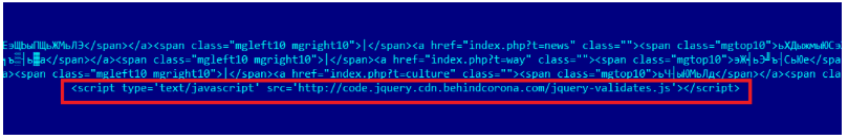
https://madnight.github.io/githut/#/pull_requests/2020/1

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],
    - ...

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]

JavaScript is complex

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

JavaScript is complex

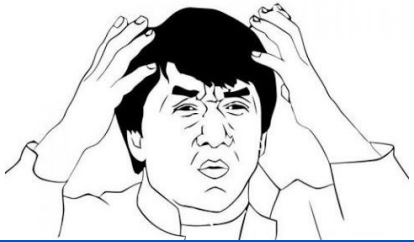
- Multi-Paradigm;
- Objects as:
  - Literals;
  - Object Factories;
  - Functions;
- Prototype-based;
  - Complex behaviours;
  - Shared behaviours.
- Resiliancy to errors, *i.e.*
  - complex syntax rules + type conversion semantics
  - redefination of global methods

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

JavaScript is complex

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

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



JavaScript is complex

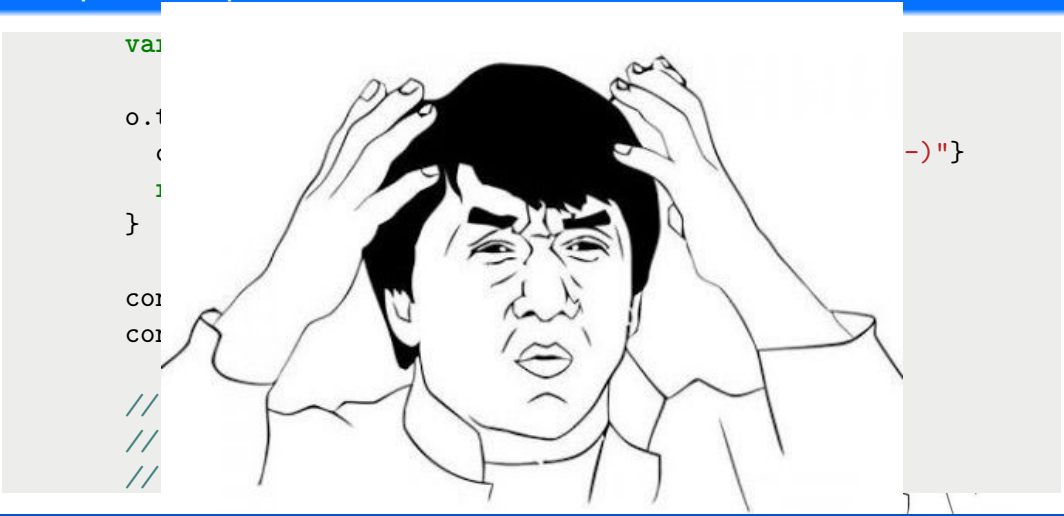
```
var o = {}

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 has a specification

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

So... Why?

"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****; ..."*

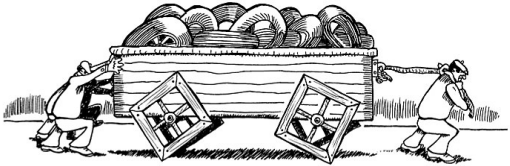
But..

This hasn't been already done by someone else?

But..

This hasn't been already done by someone else?

YES, but...



$\mathbb{K}_{JS}$ [4]

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

$\sim$ 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?

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?

λ_{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 $\mathbb{K}_{JS}$, JSCert, and λ_{JS}

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

The big common "-" of $\mathbb{K}_{JS}$, JSCert, and λ_{JS}

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

Possible solution: Decouple the semantics from its use

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

while Big-Step Rule

$$\frac{\sigma, e \Downarrow b \qquad b = true \qquad \sigma, t \Downarrow \sigma'' \qquad \sigma'', while\ e\ t \Downarrow \sigma'}{\sigma, while\ e\ t \Downarrow \sigma'}$$

$$\frac{\sigma, e \Downarrow b \qquad b = false}{\sigma, while\ e\ t \Downarrow \sigma}$$

while in skel

```
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

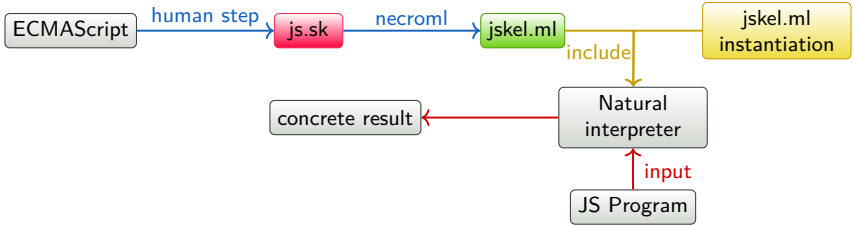
$$
\begin{array}{rcll}
\textsc{Term} & t & ::= & x_i \mid C\ t \mid (t, \ldots, t) \mid \lambda x : \tau \cdot S \\
\textsc{Skeleton} & S & ::= & x_i\ t_1 \ldots t_n \mid \mathsf{let}\ p = S\ \mathsf{in}\ S \mid (S..S) \mid t \\
\textsc{Pattern} & p & ::= & x_i \mid \_ \mid C\ p \mid (p, \ldots, p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau)
\end{array}
$$

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`, ...



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))

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))

Implicit state operations
Error-Handlers
Partial choices
Control-Flow Breaks
Loosy types
Local changes
Access to Data Structures

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 = getThisValue(v) in
            let%q v' = o_Get(base, name, thisVal) in ret<value> v'
     or     isPropertyReference v;%f 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
```

State monad

- It is a Record;
- ES environment:
  - $\mathbb{M}_\alpha$: location $\rightarrow \alpha$, $\alpha \in \{$`ExecutionContext`, `EnvironmentRecord`, `Object`, `ScriptRecord`, `ModuleRecord`, `RealmRecord`$\}$;
  - `ExecutionContext` Stack.
- It is a regular state monad;

```
type state (*left unspecified*)
type st<α> := state → (α, state)

term st_bind<α,β>: (v: st<α>) → (f: α → st<β>) → st<β> =
  λ s: state →
   let (v', s') = v s in
   f v' s'

term st_ret<α>: (v: α) → st<α> =
  λ s :state → (v, s)
```

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))

Implicit state operations

Error-Handlers

the out type

Described by ECMA	Proscribed by ECMA
`returnIfAbrupt` ?	`Assert` !

Morale:

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

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

```
type anomaly =
| AbruptAnomaly completionRecord<()>
| StringAnomaly string
| NotImplemented
```

$$\texttt{term}\ (?|!)_{<\alpha,\beta>} : \texttt{st<out<}\alpha\texttt{>>} \rightarrow (\alpha \rightarrow \texttt{st<out<}\beta\texttt{>>}) \rightarrow \texttt{st<out<}\beta\texttt{>>}$$

The Completion Record and the Error-Handling shorthands

Field Name	Value
[[Type]]	One of `normal`, `break`, `continue`, `return`, or `throw`
[[Value]]	Any **ECMAScript language value** or `empty`
[[Target]]	Any **ECMAScript string** or `empty`

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

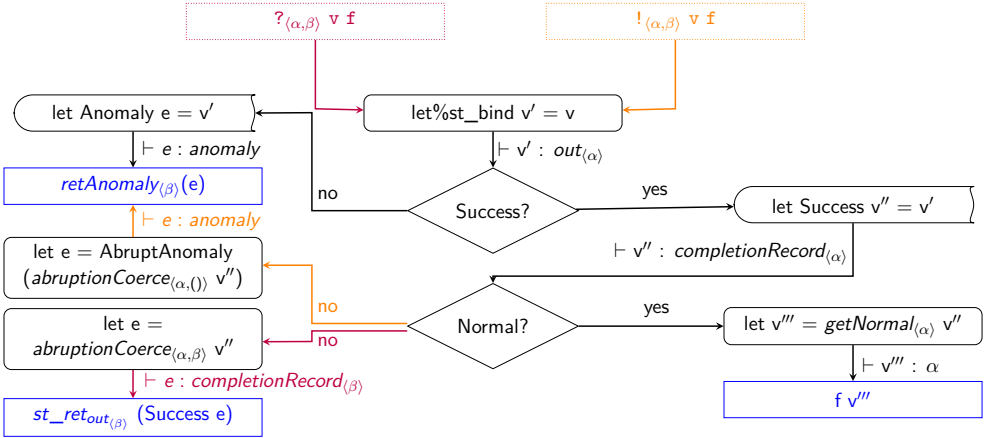
```
type completionValue<α> =
| Ok α
| Abruption maybeEmpty<value>

type completionTarget :=
    maybeEmpty<string>
```

```
type completionRecord<α> = (_Type_ : completionType,
                            _Value_ : completionValue<α>,
                            _Target_ : completionTarget)
```

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

? and ! as monadic binders



term (?|!)$_{<\alpha,\beta>}$: st<out<α>> → (α → st<out<β>>) → st<out<β>>

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))

Partial choices
Control-Flow Breaks
Local changes

Control-flow monad - Example

1. If foo is true Return 1
2. If bar is true Return 2
3. If baz is true Return 3
4. Return 4

```
(*Repetition of conditions*)
branch foo;%t 1 (*1*)
or     foo;%f bar;%t 2 (*2*)
or     foo;%f bar;%f baz;%t 3 (*3*)
or     foo;%f bar;%f baz;%f 4 (*4*)
end
```

```
(*Naive implementation*)
branch foo;%t 1 (*1*)
or     bar;%t 2 (*2*)
or     baz;%t 3 (*3*)
or     foo;%f bar;%f baz;%f 4 (*4*)
end
```

```
(*Going in deepth*)
branch foo;%t 1 (*1*)
or     foo;%f
       branch bar;%t 2 (*2*)
       or     bar;%f
              branch baz;%t 3 (*3*)
              or     baz;%f 4 (*4*)
              end
       end
end
```

term (t<α>|f<α>) : boolean → (boolean → α) → α

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> (*alluminium*)
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
```

1. If foo is true Return 1
2. If bar is true Return 2
3. If baz is true Return 3
4. Return 4

```
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
```

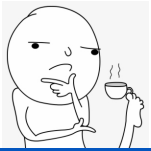
Control-Flow monad

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

```
term cf_cont<β> : () -> controlFlow<(),β> (*green*)
term cf_ret<β> : β → controlFlow<(),β> (*alluminium*)
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
```

1. If foo is true Return 1
2. If bar is true Return 2
3. If baz is true Return 3
4. Return 4



```
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
```

The or branch is verbose!

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 Breaks

Composition

What if we want to add error informations while we are dealing with the control-flow?

throw

1. If foo is true throw FooError.
2. If bar is true Return 2.
3. If baz is true Return 3.
4. Return 4.

```
let%cf_out result =
(*1*) branch foo;%p_T let%throw fe=fooError<int> in fe end;%bind
(*2*) branch bar;%p_T ret<int> 2 end;%bind
(*3*) branch bar;%p_T ret<int> 3 end;%bind
(*4*) ret<int> 4
in result
```

```
term cf_out<β,γ> : st<controlFlow<(),β>> → (β → γ) → st<γ>
term bind<α,β,γ> : st<controlFlow<α,out<β>>> → (α → st<controlFlow<γ, out<β>>>) → st<controlFlow<γ, out<β>>>
term cont<β> : () → st<controlFlow<(), out<β>>> (* b-y-g *)
term ret<β> : β → st<controlFlow<(), out<β>>> (* b-y-g*-a *)
term throw<β> : (() → st<out<β>>) → (st<controlFlow<(),out<β>>> → st<controlFlow<(),out<β>>>) →
                st<controlFlow<(),out<β>>> (* b-y-g*-a *)
term assign<α,β> : α → st<controlFlow<α,out<β>>> (* b-y-g*-o *)
term (p_T<α>|p_F<α>) : boolean → (boolean →  st<controlFlow<(),out<α>>>) → st<controlFlow<(),out<α>>> (*partiality*)
term fooError<α> : () → st<out<α>>
```

1-to-1 comparison

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

- 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.
  - Return ? base.GetBindingValueV.[[ReferencedName]], V.[[Strict]]).

```
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
  branch 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 = 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
```

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

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..

Conclusions

First year:

- Iteratively formalize JavaScript semantics;
- Defining a complex environment that semplifies writing JavaScript semantics;
- Design some of the *necro* features in order to simplify the ongoing formalization.

Publications:

- A. Khayam, L. Noizet, A. Schmitt, *"JSkel: Towards a Formalization of JavaScript's Semantics"*, *Journées Francophones des Langages Applicatifs*, 2021

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.

Then...

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.

References I

[1] D. F. Some, "Empoweb: Empowering web applications with browser extensions," *IEEE Symposium on Security Privacy, May 2019 (IEEE SP 2019)*, pp. 227–245, 05 2019.

[2] securelist.com, "Chrome 0-day exploit cve-2019-13720 used in operation wizardopium." `https://securelist.com/chrome-0-day-exploit-cve-2019-13720-used-in-operation-wizardopium/94866/`.

Last modification: 2019-11-01.

[3] G. Rosu, "K: A rewriting-based framework for computations – preliminary version –," 2007.

[4] D. Park, A. Stefănescu, and G. Roșu, "Kjs: A complete formal semantics of javascript," *ACM SIGPLAN Notices*, vol. 50, pp. 346–356, 06 2015.

[5] A. Charguéraud, "Pretty-big-step semantics," in *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, pp. 41–60, Springer, 2013.

[6] A. Charguéraud, A. Schmitt, and T. Wood, "Jsexplain: A double debugger for javascript," in *WWW '18: Companion Proceedings of the The Web Conference 2018*, pp. 691–699, 04 2018.

[7] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. 49, pp. 87–100, 01 2014.

References II

[8] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," *ECOOP, LecTure Notes in Computer Science*, pp. 126–150, 06 2010.

[9] M. Bodin, P. Gardner, T. Jensen, and A. Schmitt, "Skeletal semantics and their interpretations," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, 01 2019.

[10] E. C. Martin Bodin, Nathanael Courant and L. Noizet, "Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml."