Defensive JS in JSCert

Initial steps

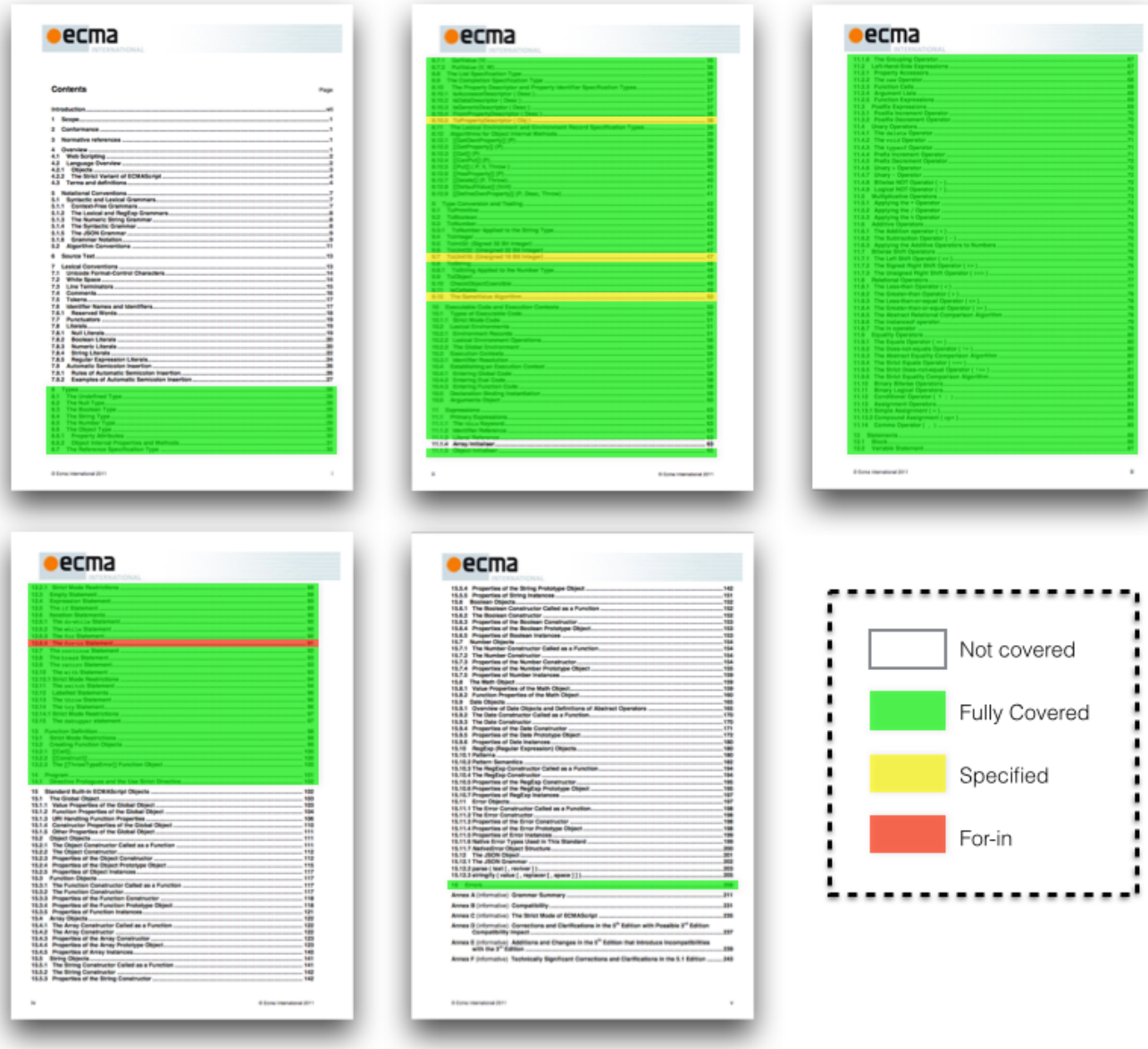
Petar Maksimović

Inria Rennes – Bretagne Atlantique

ANR AJACS Meeting
Paris, 27/11/2015

Current State of JSCert

Coverage of JSCert @ POPL'14



Coverage of JSCert @ POPL'14

- Main functionalities (Ch 8-14) dominantly implemented
  - For-in broken, ergo not implemented
  - Functionalities depending on for-in not implemented

- Libraries (Function, Array, String... (Ch 15)) not formalised
  - Functionalities depending on Ch 15 not formalised
  - Array initialiser not formalised

How to improve JSCert?

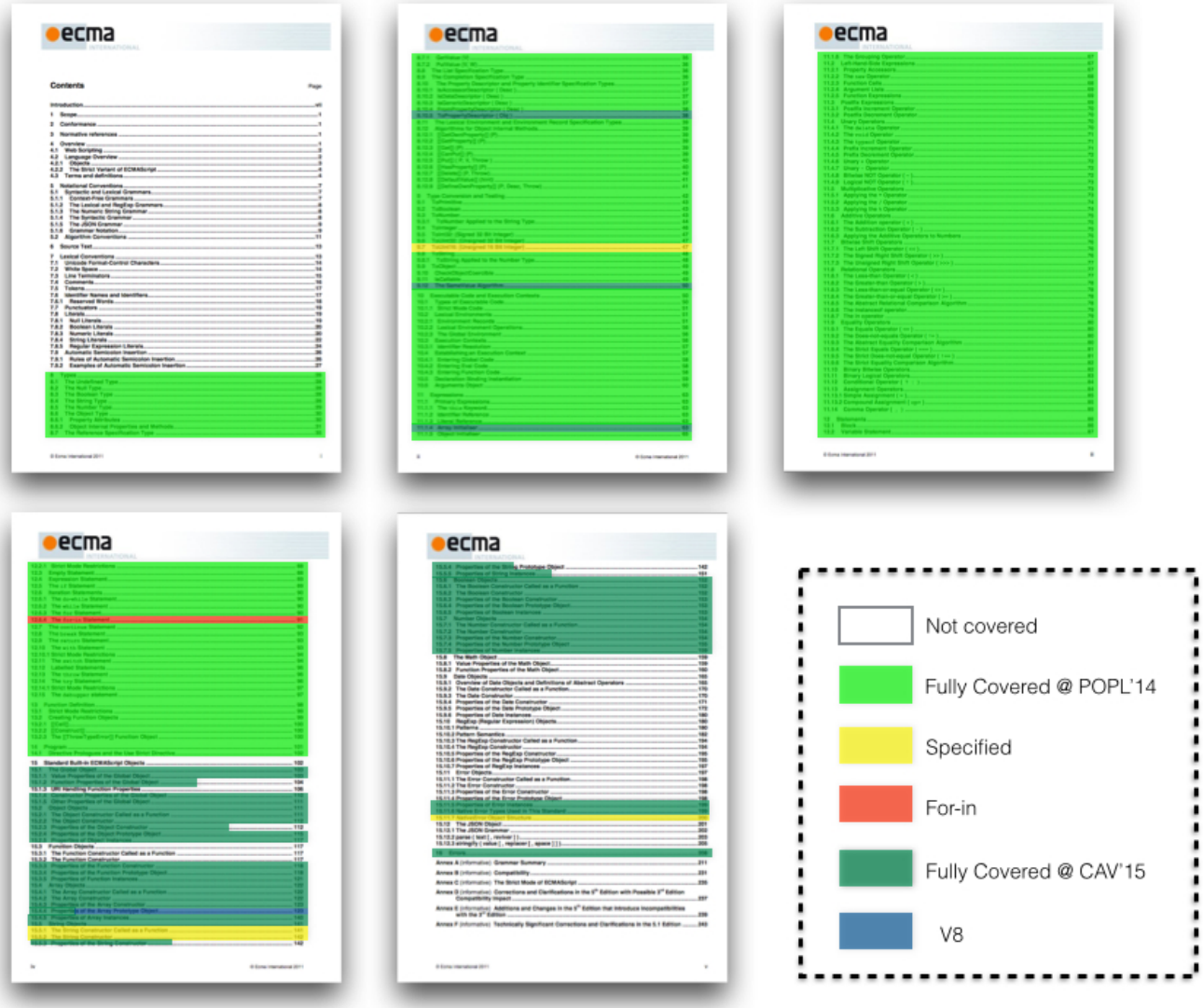
- Better Coverage of the Standard Libraries

  We need Ch 15 to fully test Ch 8–14 using Test262
- Better Testing Infrastructure

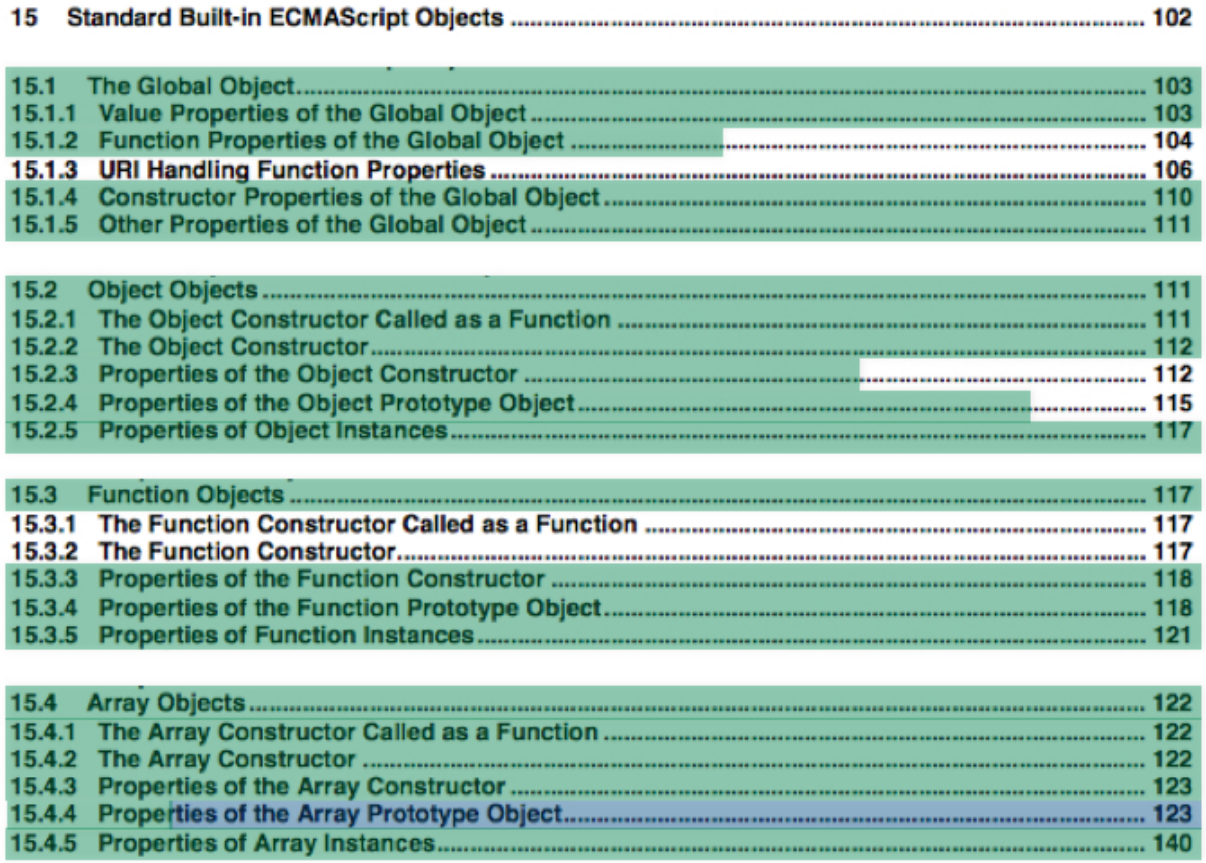
  We need to increase our confidence in the test results

- Separate *core* from *non-core* library functionality
  - *Core functionality:* everything the underlying implementation must provide
  - *Non-Core functionality:* everything that can be implemented in JavaScript on top of core functionality
- Extend JSCert, JSRef, proof with core library functionality
- Implement (or borrow an implementation of) non-core functionality in JavaScript and load it into the initial heap

Coverage of JSCert



Coverage of JSCert - Detailed Results



Global Object

29/32

Object Object

19/24

Function Objects

13/15

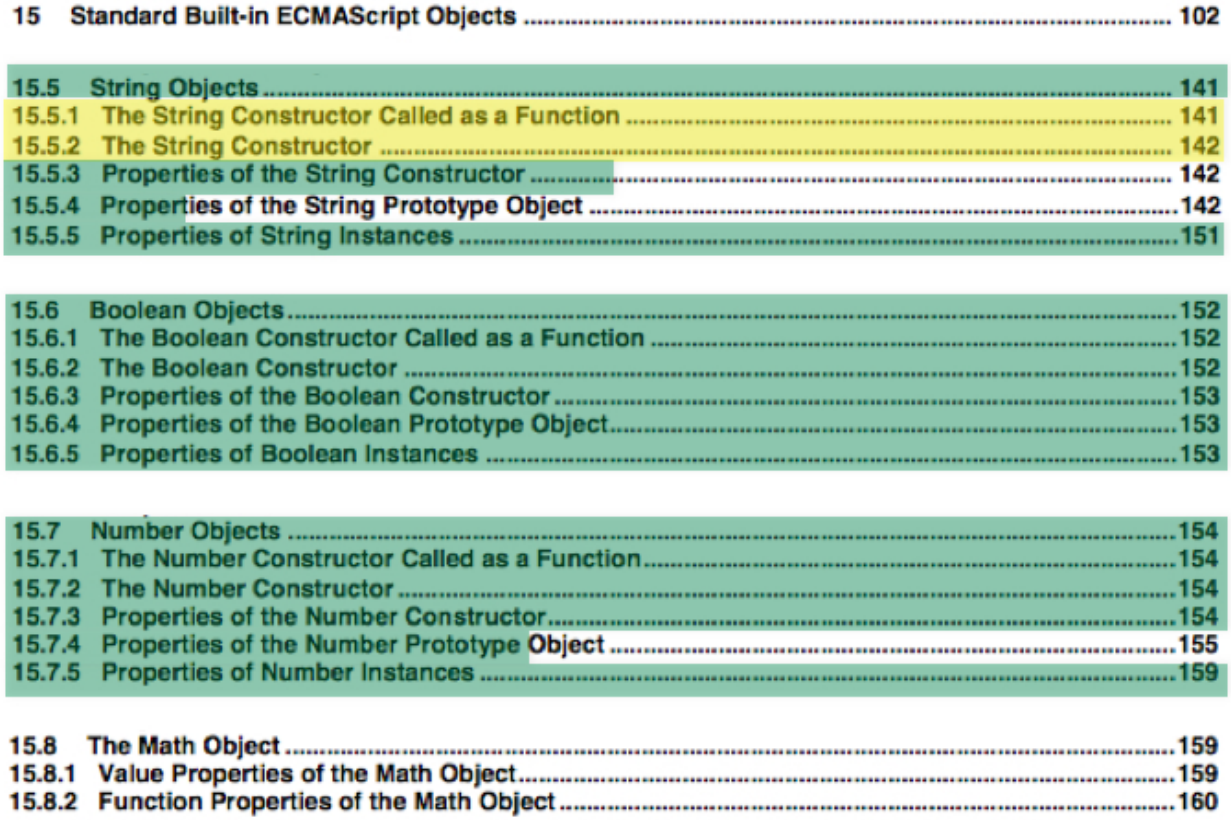
Array Objects

11/29

18/29

V8 Fully Covered @ CAV'15

Coverage of JSCert - Detailed Results



String Objects

5/26

2/26

Boolean Objects

7/7

Number Objects

12/16

Math Object

Fully Covered @ CAV'15

Specified

Coverage of JSCert - Detailed Results

Date Objects

Regular Expressions

Error Objects

12/25

11/25

Fully Covered @ CAV'15

Specified

Coverage of JSCert - Array Library

Fully Covered @ CAV'15 - **11 sections out of 29**

V8 - **18 sections out of 29**

JSRef Testing Results: Array Library

	Chs 8-14			Array Library		
	Pass	Fail	Abort	Pass	Fail	Abort
POPL'14 paper	1796	404	582	139	873	1307
POPL'14 talk	1851	392	539	149	864	1306
CAV'15 paper	2437	129	216	180	1204	935
+V8 Array	2440	126	216	1309	59	951
CAV'15 talk	2506	47	229	267	1956	69
+V8 Array	2510	43	229	2170	12	111

Evaluation

- Implementation of new features uncovered bugs in previously unused and unverified code.
- Changing the parser forced re-evaluation of parsing failures

Defensive JavaScript

What is DJS and why was it created?

- Typed subset of JavaScript
- Authors: K. Bhargavan, A. Delignat-Lavaud, S. Maffeis
- Motivation: Guarantee that program functionalities cannot be tampered with, even in a malicious environment
- Applications: Crypto libraries, single sign-on widgets, bookmarklets

What is meant by “defensiveness”?

Isolate security-critical code from the environment:

- Use function closures/wrappers
- Do not call external function explicitly
- Prevent triggering of coercions
- Prevent prototype lookups

How is this achieved?

Using a static type system:

- Static scopes
  - Limited occurrences of variable declarations,
  - Strong scoping restriction on `with`
- Statically typed objects, functions, arrays
  - No out-of-bounds, not extensible
- Coercion-free operations
- Disjoint heaps
  - No heap references allowed to be imported or exported, only `string` $\rightarrow$ `string`

Syntactic categories

- Literals (bool, number, string, object, array)
- Most unary and binary operators
- LHS-expressions (severely limiting dynamic access)
  - Properties only via `e.x`
  - Arrays and strings indexed only within bounds
- Expressions
  - Assignments; unary, binary operations
  - Fully applied function and method applications
- Statements
  - If-then-else, while, sequence
  - `with(e)s`, all `FV(s)` are properties of `e`
  - No variable declarations

Syntactic categories

- Functions
  - Variable declarations
  - Series of statements
  - Single `return` statement

$$f := \texttt{function}(x_1, \ldots, x_n)\{$$
$$\texttt{var}\ y_1 = e_1, \ldots y_m = e_m;$$
$$s_1; \ldots; s_k;\ \texttt{return}\ e\}$$

Syntactic categories

- Programs
  - Wrapper around a single function f
  - Ensures argument is string
  - Calls function, returns result

$$p_f := (\texttt{function}\,()\{\;\texttt{var}\;\_ = f;\;\texttt{return function}(x)\{\texttt{if}\;(\texttt{typeof}(x) == \text{"}string\text{"})\;\texttt{return}\;\_(x);\}\})();$$

- Wrapping—no leaking of source code of f
- Argument type check—no import of external heap refs

Types and Environments

Types and Environments

$\tau ::=$	Types
$\quad$ number $\mid$ boolean $\mid$ string $\mid$ `undefined`	$\quad$ Base Types
$\quad \rho$	$\quad$ Object
$\quad [\tau]_n$	$\quad$ Array of length $n \geq 0$
$\quad (\tau_1, \ldots, \tau_n) \to \tau_e$	$\quad$ Function $n \geq 0$
$\quad (\tau_1, \ldots, \tau_n)[\rho] \to \tau_r$	$\quad$ Method on object of type ρ $(n \geq 0)$
$\rho ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	Object Type $(n \geq 0)$
$\kappa ::=$ f $\mid$ w	Scope frame kind: `function` or `with`
$\Gamma ::= \varepsilon \mid \Gamma, [\rho]_\kappa$	Typing Environment

Illustration of the typing rules

$$\text{StrCast}\ \frac{\Gamma \vdash e : \mathsf{number}}{\Gamma \vdash e + \texttt{""} : \mathsf{string}}$$

$$\text{Array}\ \frac{\Gamma \vdash e_i : \tau \quad i \in [1..n]}{\Gamma \vdash [e_1, \ldots, e_n] : [\tau]_n} \qquad \text{ConstantIndex}\ \frac{\Gamma \vdash e : [\tau]_m \qquad m > \eta \geq 0}{\Gamma \vdash e[\eta] : \tau}$$

$$\text{VarLocal}\ \frac{\Phi(x) = \tau}{\Gamma, [\Phi]_\kappa \vdash x : \tau} \qquad \text{VarFunctionScope}\ \frac{x \notin dom(\Phi) \qquad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_\mathsf{f} \vdash x : \tau}$$

$$\text{FunctionDef}\ \frac{\begin{array}{c}\rho_k = \{(x_i : \tau_i)_{i \in [1..n]}, (y_j : \mu_j)_{j \in [1..k]}\} \\ \Gamma, [\rho_{k-1}]_\mathsf{f} \vdash d_k : \mu_k \quad k \in [1..m] \\ \Gamma, [\rho_m]_\mathsf{f} \vdash s : \texttt{undefined} \qquad \Gamma, [\rho_m]_\mathsf{f} \vdash e_r : \tau_r\end{array}}{\Gamma \vdash \texttt{function}(\tilde{x})\{\texttt{var}\ y_1\texttt{=}d_1, \ldots, y_m\texttt{=}d_m;\ s\ ;\ \texttt{return}\ e_r\} : \tilde{\tau} \to \tau_r}$$

Key Properties of DJS

Independence: p_f preserves the independence of f if any two sequences of calls with the same arguments to the result of p_f, interleaved with arbitrary JS code, return the same sequences of return values, provided no call triggered an exception.

Encapsulation: p_f encapsulates f over $\mathcal{D}$ if no JS program that runs p_f can distinguish between running p_f and $p_{f'}$ without calling the returned wrapped functions. Moreover, for any tuple $\tilde{v} \in \mathcal{D}$, heaps resulting from $p_f(\tilde{v})$ and $f(\tilde{v})$ are equivalent.

Defensiveness: If $\vdash f : \mathit{string} \to \mathit{string}$, then the DJS program p_f encapsulates f over strings and preserves its independence.

What has been done so far in Coq?

1. Complete Syntax
2. Complete Type System
3. Mapping DJS $\rightarrow$ JSCert
4. Predicate describing allowed JSCert terms
5. Proof that (3) and (4) are equivalent

Some more details

- Defined size for DJS types, DJS terms, JSCert terms
  - All proofs by induction on appropriate size
  - Using size, derive structural induction principles
- Rules of variable length (object, array, function)
  - Using Forall from TLC
  - Pretty-big-step style rules
  - PBS for functions, Forall for the rest

```
| typing_Array : forall G t le,
    length le > 0 ->
    Forall (fun e => G |- e :: t) le ->
    G |- (djs_a le) :: [[t, length le]]
```

Some more details

- Translation DJS → JSCert (`toJSC`)
  - Mapping of unary and binary operators
  - Follow descriptions from the syntax

```
(* Constant Array Index *)
 | djs_cai  ε n ⇒ expr_access (toJSC_ε ε)
                              (expr_literal (literal_number (JsNumber.of_int n)))

(* Functions *)
with toJSC_ϕ ϕ :=
match ϕ with
 | djs_ϕ_main lx τρy s ε ⇒
       expr_function
         None (* Functions have no names *)
         lx   (* List of variables       *)
         (* Function body *)
         (funcbody_intro
            (prog_intro
               true (* Body is always strict *)
               (
                 (* Variables *) element_stat (stat_var_decl
                                                 ((fix var_decl τρy :=
                                                     match τρy with
                                                       | nil ⇒ nil
                                                       | (y, δε) :: τρy ⇒ (y, Some (toJSC_δε δε)) :: var_decl τρy
                                                     end) τρy)) ::
                 (* Statement *) element_stat (toJSC_stat s) ::
                 (* Return    *) element_stat (stat_return (Some (toJSC_ε ε))) :: nil
               )
            )
            "function() {}"
         )
end
```

Some more details

- Allowed terms in JSCert (`DJS_allowed_term`)
  - Mapping of operators
  - Follow descriptions from the syntax

```
(* Functions do not have names, programs always in strict mode *)
| expr_function None lx (funcbody_intro (prog_intro true lel) msg) =>
  msg = "function() {}" /\
  match lel with
   (* first statement is a variable declaration *)
   | element_stat (stat_var_decl τpy) :: lel =>
     ((fix DJS_allowed_lje τpy :=
         match τpy with
           | nil => True
           | (_, Some je) :: τpy => DJS_allowed_expr je /\ DJS_allowed_lje τpy
           | _ => False
         end) τpy) /\
     match lel with
       (* functions = statement, then return *)
       | element_stat s :: element_stat (stat_return (Some je)) :: nil =>
           DJS_allowed_stat s /\ DJS_allowed_expr je /\ not_a_function je
       | _ => False
     end
   | _ => False
  end
```

Some more details

- Relationship between `toJSC` and `DJS_allowed_term`
  - `T` – DJS term, `t` – JSCert term

$$\forall \texttt{t}, \texttt{DJS\_allowed\_term t} \leftrightarrow (\exists \texttt{T}, \texttt{toJSC T} = \texttt{t})$$

What are the next steps?

- Compatibility between Γ in DJS and (S, C) in JSCert
- **Type safety**
- Properties of the heap
  - Equivalence of heaps
  - Separation
- Additional information within JSCert
  - Traces?
- Semantics of DJS?

Thank you for your attention!