Trusted JavaScript Semantics

Philippa Gardner

http://jscert.org

Imperial College London

UK Research Institute for Automatic Program Analysis and Verification, funded by GCHQ with EPSRC

People

INRIA

- Martin Bodin
- Arthur Charguéraud
- Alan Schmitt

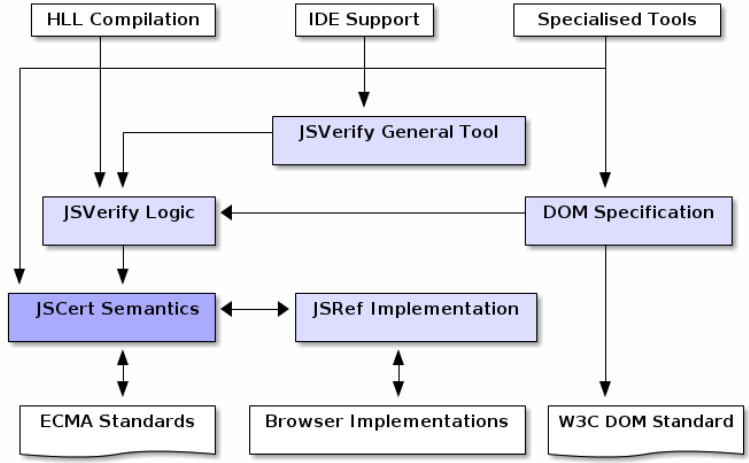
Imperial College

- Daniele Filaretti
- Philippa Gardner
- Sergio Maffeis
- Daiva Naudžiūnienė
- Gareth Smith
- Adam Wright

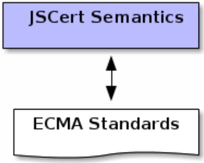
JavaScript Semantics

- Initial Implementation (Netscape Navigator 1996)
- ECMAScript 3 standard (1999)
- Formal semantics for the full language (APLAS'08)
- ECMAScript 5 (2009)
- λ_{JS}: semantics via translation into a λ-calculus with references (ECOOP'10)
- Program logic for a core part of the language (POPL'12)
- λS5: like λ_{JS} for ES5 strict mode (DLS'12)
- F^* to JavaScript, a full abstraction result (POPL'13)

The Big Picture

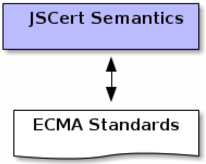


JSCert



- A Coq specification of the ES5 standard (strict and non-strict)
- Eyeball-closeness to ES5 standard
- Safety properties (provided that we trust Coq):
  - no well-formed program ‘gets stuck’;
  - the heap is always well-formed.

JSCert progress



JSCert Semantics

ECMA Standards

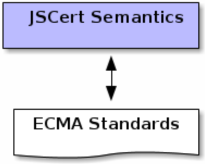
Subset of JavaScript formalized so far:

- variables: scopes, prototype chains, assignment
- functions: declare, call, new
- objects: delete, access, get, set
- operators: unary and binary (most useful ones)
- control flow: sequence, conditional, while loop, if, break, continue, etc
- with construct, this construct
- exceptions: throw, try-catch-finally
- type conversions
- eval (on-going, parameterised by any trusted parser)

Main missing features:

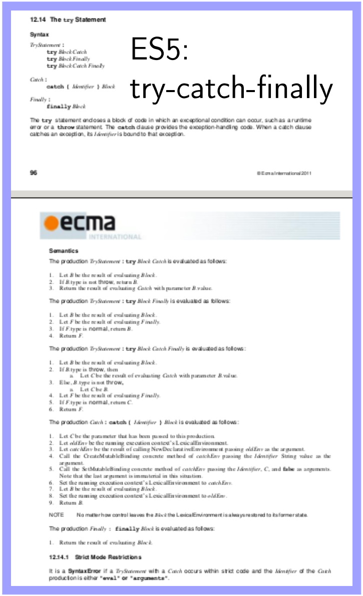
- control flow: switch (simple), for loops (interesting, on-going)
- parsing (affects eval)
- extensions: arrays, regexp, errors, . . .

Direct ‘eyeball’ correspondence



ES5:

try-catch-finally



12.14 The `try` Statement

Syntax

TryStatement :
    `try` *Block Catch*
    `try` *Block Finally*
    `try` *Block Catch Finally*

Catch :
    `catch` **`(`** *Identifier* **`)`** *Block*

Finally :
    `finally` *Block*

The **`try`** statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a **`throw`** statement. The **`catch`** clause provides the exception-handling code. When a catch clause catches an exception, its *Identifier* is bound to that exception.

96 © Ecma International 2011

ecma INTERNATIONAL

Semantics

The production *TryStatement* : **`try`** *Block Catch* is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is not throw, return *B*.
3. Return the result of evaluating *Catch* with parameter *B*.value.

The production *TryStatement* : **`try`** *Block Finally* is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.type is normal, return *B*.
4. Return *F*.

The production *TryStatement* : **`try`** *Block Catch Finally* is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is throw, then
    a. Let *C* be the result of evaluating *Catch* with parameter *B*.value.
3. Else, *B*.type is not throw,
    a. Let *C* be *B*.
4. Let *F* be the result of evaluating *Finally*.
5. If *F*.type is normal, return *C*.
6. Return *F*.

The production *Catch* : **`catch`** **`(`** *Identifier* **`)`** *Block* is evaluated as follows:

1. Let *C* be the parameter that has been passed to this production.
2. Let *oldEnv* be the running execution context's LexicalEnvironment.
3. Let *catchEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
4. Call the CreateMutableBinding concrete method of *catchEnv* passing the *Identifier* String value as the argument.
5. Call the SetMutableBinding concrete method of *catchEnv* passing the *Identifier*, *C*, and **false** as arguments. Note that the last argument is immaterial in this situation.
6. Set the running execution context's LexicalEnvironment to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.
8. Set the running execution context's LexicalEnvironment to *oldEnv*.
9. Return *B*.

NOTE No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

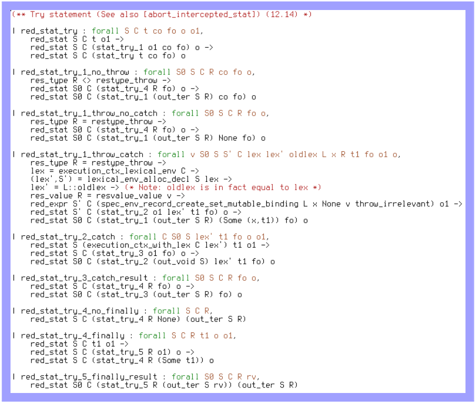
The production *Finally* : **`finally`** *Block* is evaluated as follows:

1. Return the result of evaluating *Block*.

12.14.1 Strict Mode Restrictions

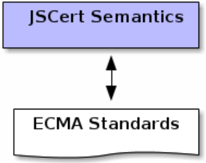
It is a **SyntaxError** if a *TryStatement* with a *Catch* occurs within strict code and the *Identifier* of the *Catch* production is either **`"eval"`** or **`"arguments"`**.

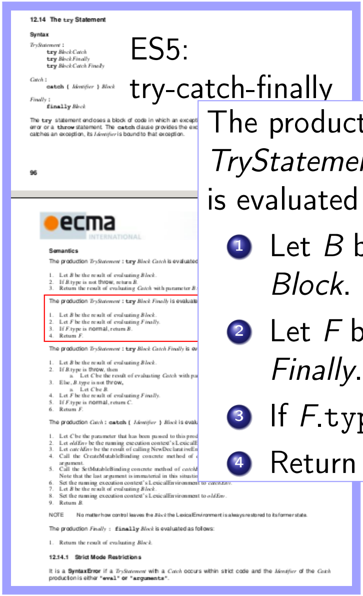
JSCert: try-catch-finally



```
(** Try statement (See also [abort_intercepted_stat]) (12.14) *)

| red_stat_try : forall S C t co fo o o1,
    red_stat S C t o1 ->
    red_stat S C (stat_try_1 o1 co fo) o ->
    red_stat S C (stat_try t co fo) o

| red_stat_try_1_no_throw : forall S0 S C R co fo o,
    res_type R <> restype_throw ->
    red_stat S0 C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) co fo) o

| red_stat_try_1_throw_no_catch : forall S0 S C R fo o,
    res_type R = restype_throw ->
    red_stat S0 C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) None fo) o

| red_stat_try_1_throw_catch : forall v S0 S S' C lex lex' oldlex L x R t1 fo o1 o,
    res_type R = restype_throw ->
    lex = execution_ctx_lexical_env C ->
    (lex',S') = lexical_env_alloc_decl S lex ->
    lex' = L::oldlex -> (* Note: oldlex is in fact equal to lex *)
    res_value R = resvalue_value v ->
    red_expr S' C (spec_env_record_create_set_mutable_binding L x None v throw_irrelevant) o1 ->
    red_stat S' C (stat_try_2 o1 lex' t1 fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) (Some (x,t1)) fo) o

| red_stat_try_2_catch : forall C S0 S lex' t1 fo o o1,
    red_stat S (execution_ctx_with_lex C lex') t1 o1 ->
    red_stat S C (stat_try_3 o1 fo) o ->
    red_stat S0 C (stat_try_2 (out_void S) lex' t1 fo) o

| red_stat_try_3_catch_result : forall S0 S C R fo o,
    red_stat S C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_3 (out_ter S R) fo) o

| red_stat_try_4_no_finally : forall S C R,
    red_stat S C (stat_try_4 R None) (out_ter S R)

| red_stat_try_4_finally : forall S C R t1 o o1,
    red_stat S C t1 o1 ->
    red_stat S C (stat_try_5 R o1) o ->
    red_stat S C (stat_try_4 R (Some t1)) o

| red_stat_try_5_finally_result : forall S0 S C R rv,
    red_stat S0 C (stat_try_5 R (out_ter S rv)) (out_ter S R)
```

Direct ‘eyeball’ correspondence





12.14 The `try` Statement

Syntax

TryStatement :
  `try` *Block Catch*
  `try` *Block Finally*
  `try` *Block Catch Finally*

Catch :
  `catch` (*Identifier*) *Block*

Finally :
  `finally` *Block*

The **`try`** statement encloses a block of code in which an excepti
error or a **`throw`** statement. The **`catch`** clause provides the exc
catches an exception, its *Identifier* is bound to that exception.

96

Semantics

The production *TryStatement* : **`try`** *Block Catch* is evaluated

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is not throw, return *B*.
3. Return the result of evaluating *Catch* with parameter *B*.

The production *TryStatement* : **`try`** *Block Finally* is evaluate

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.type is normal, return *B*.
4. Return *F*.

The production *TryStatement* : **`try`** *Block Catch Finally* is ev

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is throw, then
   a. Let *C* be the result of evaluating *Catch* with pa
3. Else, *B*.type is not throw,
   a. Let *C* be *B*.
4. Let *F* be the result of evaluating *Finally*.
5. If *F*.type is normal, return *C*.
6. Return *F*.

The production *Catch* : **`catch`** (*Identifier*) *Block* is evalu

1. Let *C* be the parameter that has been passed to this prod
2. Let *oldEnv* be the running execution context's LexicalE
3. Let *catchEnv* be the result of calling NewDeclarativeEn
4. Call the CreateMutableBinding concrete method of c
   argument.
5. Call the SetMutableBinding concrete method of *catchE*
   Note that the last argument is immaterial in this situatio
6. Set the running execution context's LexicalEnvironment to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.
8. Set the running execution context's LexicalEnvironment to *oldEnv*.
9. Return *B*.

NOTE No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

The production *Finally* : **`finally`** *Block* is evaluated as follows:

1. Return the result of evaluating *Block*.

12.14.1 Strict Mode Restrictions

It is a **SyntaxError** if a *TryStatement* with a *Catch* occurs within strict code and the *Identifier* of the *Catch* production is either **`"eval"`** or **`"arguments"`**.

ES5:

try-catch-finally

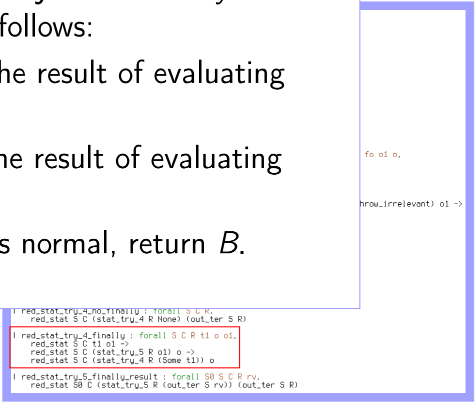
The production

TryStatement : **try** *Block Finally*

is evaluated as follows:

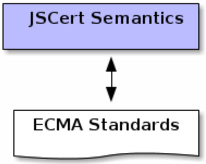
1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.`type` is normal, return *B*.
4. Return *F*.

nally



```
fo o1 o,

hrow_irrelevant) o1 ->

| red_stat_try_4_no_finally : forall S C R,
    red_stat S C (stat_try_4 R None) (out_ter S R)

| red_stat_try_4_finally : forall S C R t1 o o1,
    red_stat S C t1 o1 ->
    red_stat S C (stat_try_5 R o1) o ->
    red_stat S C (stat_try_4 R (Some t1)) o

| red_stat_try_5_finally_result : forall S0 S C R rv,
    red_stat S0 C (stat_try_5 R (out_ter S rv)) (out_ter S R)
```

What do Browsers do?



JSCert Semantics

ECMA Standards

```
try { "try" } finally { "finally" }
```

ES5, Opera: (normal, `"try"`)

Chrome, FF, IE, Safari: (normal, `"finally"`)

```
try { "try" ; break } finally { "finally" }
```

ES5, Opera, Safari: (break, `"try"`)

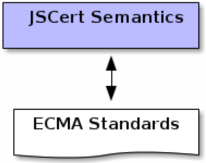
Chrome, FF, IE: (break, `"finally"`)

```
try { "try" } finally { "finally" ; break }
```

ES5, Chrome, FF, IE, Safari: (break, `"finally"`)

Opera: (break, `"try"`)

What do Browsers do?



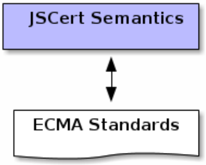
```
while(true) {
    try { "try" ; break }
    finally { "finally" }
}
```

Chrome: (break, `"finally"`)

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    y = "done"
}
```

Chrome: (break, `"try"`)

What do Browsers do?


JSCert Semantics

ECMA Standards

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    if(true) {2} else {var x = 3}
}
```

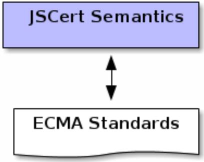
Chrome: (break, `"finally"`)

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    if(true) {2} else {3}
}
```

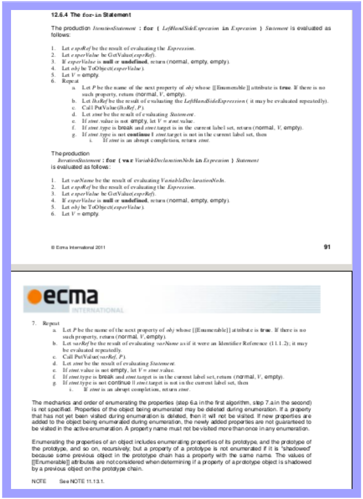
Chrome: (break, `"try"`)

More eyeballing:

ES5: for-in



```
for(var i in ob) {
    alert(i);
}
```



12.6.4 The for-in Statement

The production *IterationStatement* : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement* is evaluated as follows:

```
1. Let exprRef be the result of evaluating the Expression.
2. Let experValue be GetValue(exprRef).
3. If experValue is null or undefined, return (normal, empty, empty).
4. Let obj be ToObject(experValue).
5. Let V = empty.
6. Repeat
    a. Let P be the name of the next property of obj whose [[Enumerable]] attribute is true. If there is no
       such property, return (normal, V, empty).
    b. Let lhsRef be the result of evaluating the LeftHandSideExpression ( it may be evaluated repeatedly).
    c. Call PutValue(lhsRef, P).
    d. Let stmt be the result of evaluating Statement.
    e. If stmt.value is not empty, let V = stmt.value.
    f. If stmt.type is break and stmt.target is in the current label set, return (normal, V, empty).
    g. If stmt.type is not continue || stmt.target is not in the current label set, then
        i. If stmt is an abrupt completion, return stmt.
```

The production

IterationStatement : **for** (**var** *VariableDeclarationNoIn* **in** *Expression*) *Statement*

is evaluated as follows:

```
1. Let varName be the result of evaluating VariableDeclarationNoIn.
2. Let exprRef be the result of evaluating the Expression.
3. Let experValue be GetValue(exprRef).
4. If experValue is null or undefined, return (normal, empty, empty).
5. Let obj be ToObject(experValue).
6. Let V = empty.
```

© Ecma International 2011 91

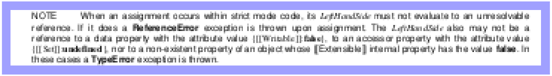
ecma INTERNATIONAL

```
7. Repeat
    a. Let P be the name of the next property of obj whose [[Enumerable]] attribute is true. If there is no
       such property, return (normal, V, empty).
    b. Let varRef be the result of evaluating varName as if it were an Identifier Reference (11.1.2); it may
       be evaluated repeatedly.
    c. Call PutValue(varRef, P).
    d. Let stmt be the result of evaluating Statement.
    e. If stmt.value is not empty, let V = stmt.value.
    f. If stmt.type is break and stmt.target is in the current label set, return (normal, V, empty).
    g. If stmt.type is not continue || stmt.target is not in the current label set, then
        i. If stmt is an abrupt completion, return stmt.
```

The mechanics and order of enumerating the properties (step 6.a in the first algorithm, step 7.a in the second) is not specified. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration.

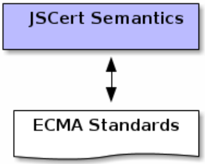
Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name. The values of [[Enumerable]] attributes are not considered when determining if a property of a prototype object is shadowed by a previous object on the prototype chain.

NOTE See NOTE 11.13.1.



NOTE When an assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable reference. If it does a **ReferenceError** exception is thrown upon assignment. The *LeftHandSide* also may not be a reference to a data property with the attribute value {[[Writable]]:**false**}, to an accessor property with the attribute value {[[Set]]:**undefined**}, nor to a non-existent property of an object whose [[Extensible]] internal property has the value **false**. In these cases a **TypeError** exception is thrown.

Key points

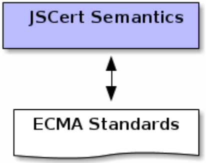


JSCert Semantics

ECMA Standards

- You can visit properties in any order.
- You can delete properties during the traversal: if they haven't been visited yet, they won't be.
- You can add properties during the traversal: they may or may not be visited.
- You must visit properties in your prototype chain.
  - . . . unless they are shadowed by another property of the same name.
- You must not visit 'non-enumerable' properties.
  - . . . even if they're shadowing enumerable ones.

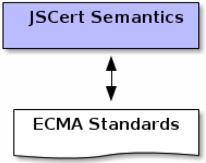
Shadowing and non-enumerable properties.



If a non-enumerable property shadows an enumerable one, neither should be visible.

Firefox	Pass, they were both hidden
Chrome	Fail, we saw the non-enumerable one
Safari	Fail, we saw the non-enumerable one
Opera	Pass, they were both hidden
IE	Fail, we saw the non-enumerable one

Shadowing, mutable state and non-enumerable properties.



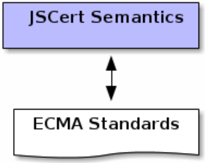
JSCert Semantics

ECMA Standards

If we delete an enumerable property which was shadowing a non-enumerable one, the non-enumerable one *still* shouldn't be visited.

Firefox	Pass, the non-enumerable property was not visited
Chrome	Fail, we saw the non-enumerable property
Safari	Fail, we saw the non-enumerable property
Opera	Fail, we saw the non-enumerable property
IE	Pass, the non-enumerable property was not visited

JSCert Summary

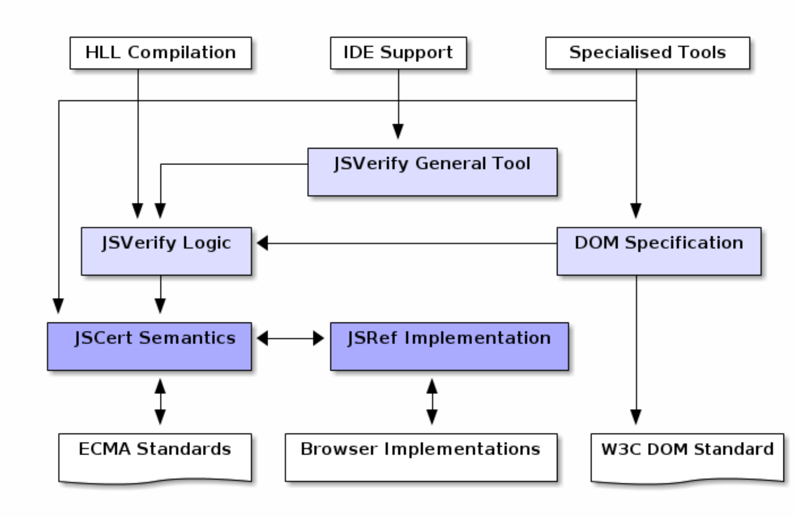


JSCert Semantics

ECMA Standards

- A Coq specification of JavaScript, nearly complete
- Eyeball-closeness to ES5 standard
- Some safety properties proved
- Bugs reported in:
  - Firefox
  - Chrome
  - Safari
  - ES6 draft standard
  - Test262 official test suite

The Big Picture



HLL Compilation

IDE Support

Specialised Tools

JSVerify General Tool

JSVerify Logic

DOM Specification

JSCert Semantics

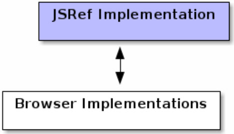
JSRef Implementation

ECMA Standards

Browser Implementations

W3C DOM Standard

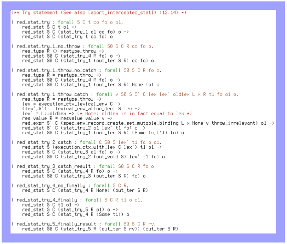
JSRef


JSRef Implementation

Browser Implementations

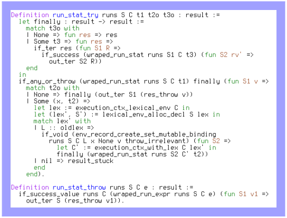
- An executable reference interpreter, starting to run now.
- Derived from JSCert semantics.
- Tested using Test262 and the Firefox test suite.

JSCert: try-catch-finally



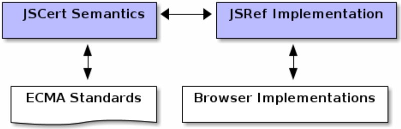
```
(** Try statement (See also [abort_intercepted_stat]) (12.14) *)

| red_stat_try : forall S C t co fo o o1,
    red_stat S C t o1 ->
    red_stat S C (stat_try_1 o1 co fo) o ->
    red_stat S C (stat_try t co fo) o

| red_stat_try_1_no_throw : forall S0 S C R co fo o,
    res_type R <> restype_throw ->
    red_stat S0 C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) co fo) o

| red_stat_try_1_throw_no_catch : forall S0 S C R fo o,
    res_type R = restype_throw ->
    red_stat S0 C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) None fo) o

| red_stat_try_1_throw_catch : forall v S0 S S' C lex lex' oldlex L x R t1 fo o1 o,
    res_type R = restype_throw ->
    lex = execution_ctx_lexical_env C ->
    (lex',S') = lexical_env_alloc_decl S lex ->
    lex' = L::oldlex -> (* Note: oldlex is in fact equal to lex *)
    res_value R = resvalue_value v ->
    red_expr S' C (spec_env_record_create_set_mutable_binding L x None v throw_irrelevant) o1 ->
    red_stat S' C (stat_try_2 o1 lex' t1 fo) o ->
    red_stat S0 C (stat_try_1 (out_ter S R) (Some (x,t1)) fo) o

| red_stat_try_2_catch : forall C S0 S lex' t1 fo o o1,
    red_stat S (execution_ctx_with_lex C lex') t1 o1 ->
    red_stat S C (stat_try_3 o1 fo) o ->
    red_stat S0 C (stat_try_2 (out_void S) lex' t1 fo) o

| red_stat_try_3_catch_result : forall S0 S C R fo o,
    red_stat S C (stat_try_4 R fo) o ->
    red_stat S0 C (stat_try_3 (out_ter S R) fo) o

| red_stat_try_4_no_finally : forall S C R,
    red_stat S C (stat_try_4 R None) (out_ter S R)

| red_stat_try_4_finally : forall S C R t1 o o1,
    red_stat S C t1 o1 ->
    red_stat S C (stat_try_5 R o1) o ->
    red_stat S C (stat_try_4 R (Some t1)) o

| red_stat_try_5_finally_result : forall S0 S C R rv,
    red_stat S0 C (stat_try_5 R (out_ter S rv)) (out_ter S R)
```

JSRef: try-catch-finally



```
Definition run_stat_try runs S C t1 t2o t3o : result :=
  let finally : result -> result :=
    match t3o with
    | None => fun res => res
    | Some t3 => fun res =>
      if_ter res (fun S1 R =>
        if_success (wraped_run_stat runs S1 C t3) (fun S2 rv' =>
          out_ter S2 R))
    end
  in
  if_any_or_throw (wraped_run_stat runs S C t1) finally (fun S1 v =>
    match t2o with
    | None => finally (out_ter S1 (res_throw v))
    | Some (x, t2) =>
      let lex := execution_ctx_lexical_env C in
      let (lex', S') := lexical_env_alloc_decl S lex in
      match lex' with
      | L :: oldlex =>
        if_void (env_record_create_set_mutable_binding
          runs S C L x None v throw_irrelevant) (fun S2 =>
            let C' := execution_ctx_with_lex C lex' in
            finally (wraped_run_stat runs S2 C' t2))
      | nil => result_stuck
      end
    end).

Definition run_stat_throw runs S C e : result :=
  if_success_value runs C (wraped_run_expr runs S C e) (fun S1 v1 =>
    out_ter S (res_throw v1)).
```

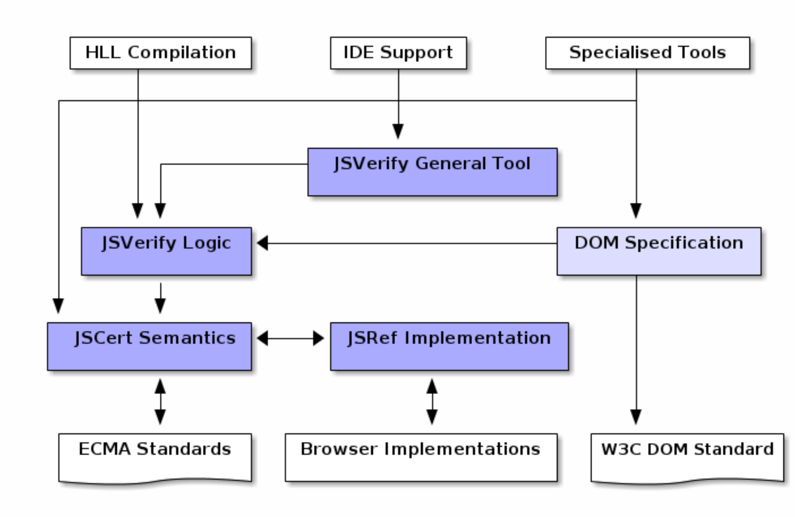
JSCert and JSRef



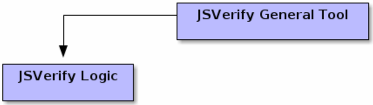
On-going tasks:

- Prove JSRef correct with respect to JSCert
- Use tests to analyse differences between JSRef and e.g. the Firefox implementation
- Add ‘faithful ES5’ and ‘Firefox’ behaviour flags to both JSCert and JSRef
- Provide more complete test coverage for ES5

The Big Picture



JSVerify



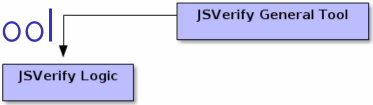
Agenda:

- Program logic for JavaScript based on separation logic.
- Abstraction: escape the gnarly details!
- Automation: interactively explore program behaviour, or even discover specifications automatically.

Progress so far:

- Core logic (POPL'12).
- Higher-order logic to come this year.
- Prototype JuS tool for automation (JSTools'13).
- Coq formalisation just starting.

JuS: A prototype verification tool


JSVerify General Tool

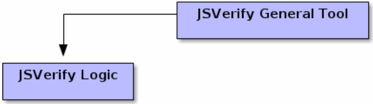
JSVerify Logic

A simple Firefox test case:

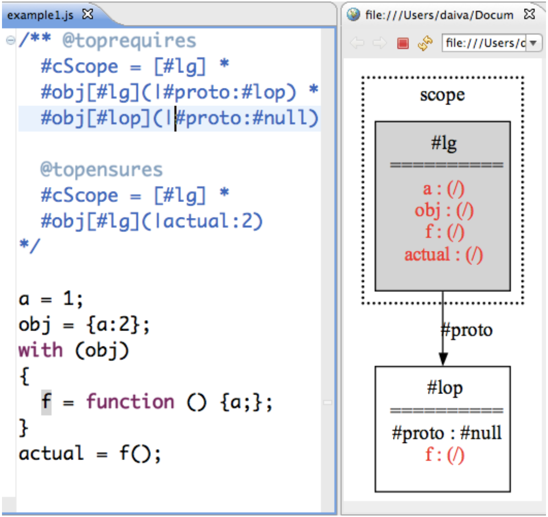
```
a = 1;
obj = {a:2};
with(obj) {
    f = function(){return a;};
}
actual = f();
```

The final value of `actual` should be 2.

A simple partially-specified starting state

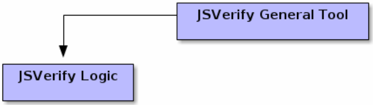


Precondition



```
/** @toprequires
   #cScope = [#lg] *
   #obj[#lg](|#proto:#lop) *
   #obj[#lop](|#proto:#null)

   @topensures
   #cScope = [#lg] *
   #obj[#lg](|actual:2)
*/

a = 1;
obj = {a:2};
with (obj)
{
   f = function () {a;};
}
actual = f();
```

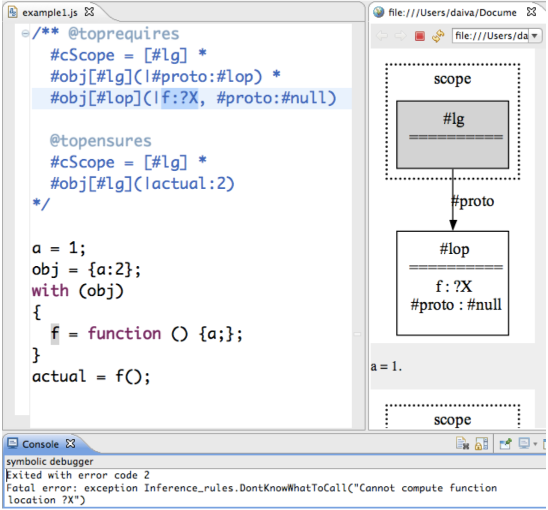
Yes we do!



JSVerify General Tool

JSVerify Logic

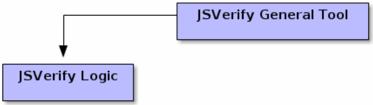
Precondition



```
/** @toprequires
   #cScope = [#lg] *
   #obj[#lg](|#proto:#lop) *
   #obj[#lop](|f:?X, #proto:#null)

   @topensures
   #cScope = [#lg] *
   #obj[#lg](|actual:2)
*/

a = 1;
obj = {a:2};
with (obj)
{
   f = function () {a;};
}
actual = f();
```

```
symbolic debugger
Exited with error code 2
Fatal error: exception Inference_rules.DontKnowWhatToCall("Cannot compute function
location ?X")
```

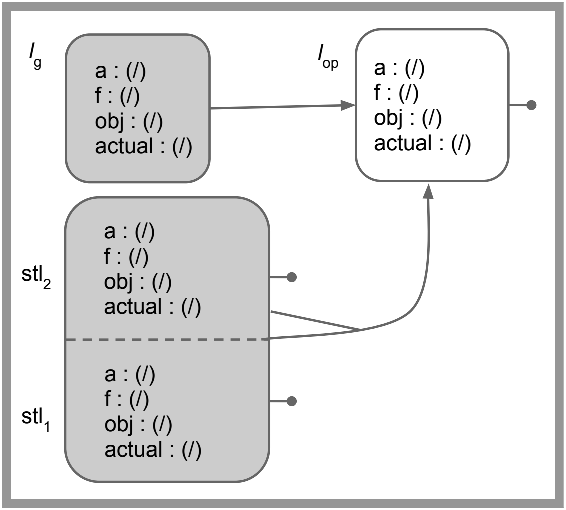
A more general precondition



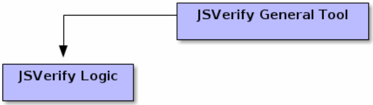
Program

```
a = 1;
obj = {a:2};
with(obj) {
    f = function(){
        return a;
    };
}
actual = f();
```

Automatically generated precondition



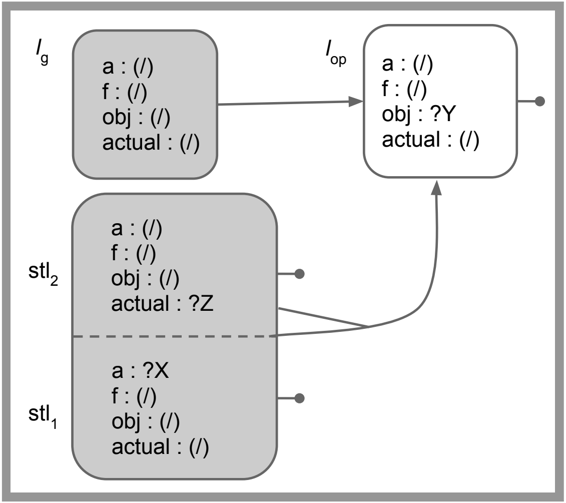
Exploring the precondition



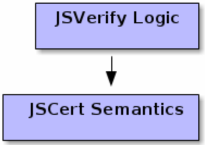
Program

```
a = 1;
obj = {a:2};
with(obj) {
    f = function(){
        return a;
    };
}
actual = f();
```

Manually edited precondition



JSVerify and JSCert



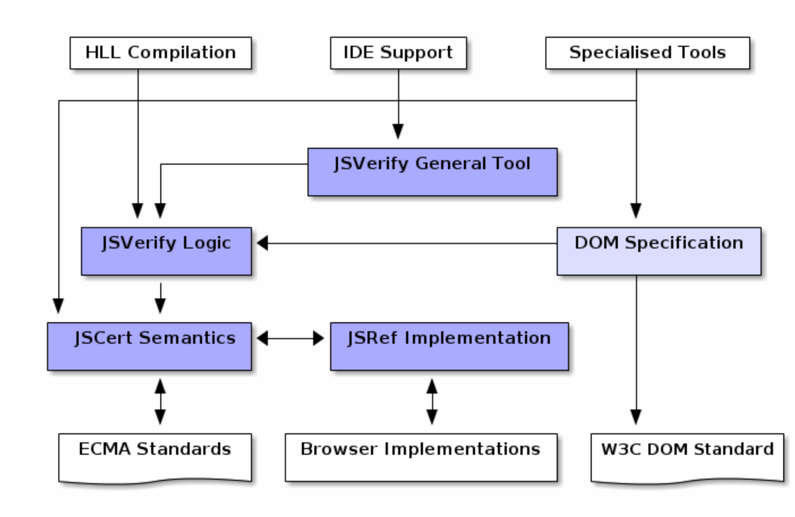
JSVerify Logic

JSCert Semantics

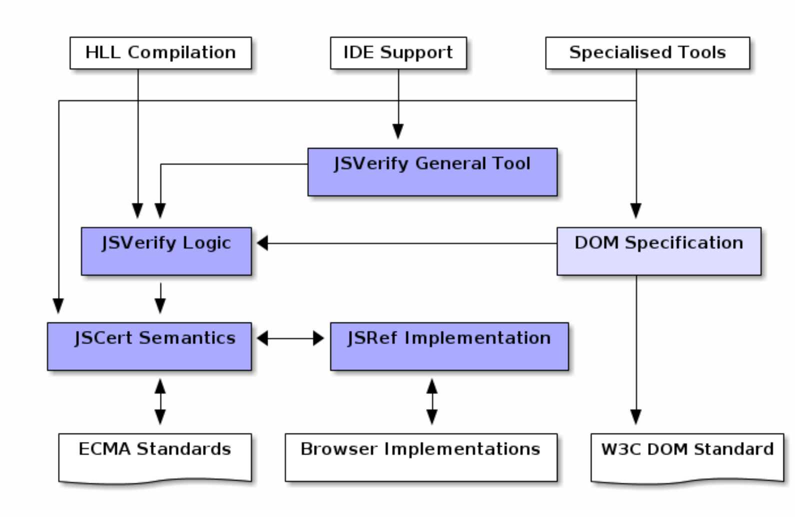
Agenda:

- JSVerify logic formalised in Coq (just begun)
- JSVerify logic proved sound with respect to JSCert semantics (both ES5 and Firefox)
- JSVerify tools able to produce Coq-checkable proof scripts

The Big Picture



Questions?


HLL Compilation
IDE Support
Specialised Tools
JSVerify General Tool
JSVerify Logic
DOM Specification
JSCert Semantics
JSRef Implementation
ECMA Standards
Browser Implementations
W3C DOM Standard