Certified desugaring of Javascript

λ_{JS} meets JSCert

Marek Materzok

1

Introduction

What is JSCert

- EcmaScript 5 formalized in Coq
- Verified and tested interpreter (JSRef)
- Interpreter extracted to OCaml

What is λ_{JS} (S5 variant)

- A core calculus for EcmaScript 5, based on the call-by-value lambda calculus
- An embedding of ES5 into the core calculus (the *desugaring function*)
- The environment file – ES5 algorithms and built-in objects implemented in the core calculus
- Interpreter of the core calculus

Usefulness of λ_{JS}

- Designing of type systems and program analysis tools
- Examples: typecheckers for Javascript (TeJaS, Strobe)
- Uses: ADsafety, typed jQuery, verified private browsing

More info: `http://www.jswebtools.org/`

LambdaCert: S5 married with JSCert

- Semantics of the S5 core calculus formalized in Coq
- The interpreter reimplemented in Coq
- Desugaring function (ES5 to S5 core) reimplemented in Coq
- Interpreter + desugaring extracted to OCaml
- Equivalence proof for the semantics and the interpreter
- Work in progress: proving soundness of S5 (core + desugaring + environment) with respect to JSCert

What's done?

A lot. What's not (yet) done?

- bit-shift operators
- for, for-in, do-while, switch
- eval
- arguments object
- function object creation ☹
- defineOwnProperty ☹
- internal methods for some exotic objects
- built-in functions (ES5 chapter 15)

2

Small example

Factorial computation

The following simple program is covered by the proofs:

```
var x=5, i=1; while(x) { i *= x-- }; i
```

Factorial computation, in λ_{JS}

```
%EnvDefineVar($vcontext, "x", false, $strict);
%EnvDefineVar($vcontext, "i", false, $strict);
%EnvAssign($context, "x", func() {5.}, $strict);
%EnvAssign($context, "i", func() {1.}, $strict); empty;;
label %brk: {
  rec (#while_loop = func() {
    if (%ToBoolean(%EnvGet($context, "x", $strict))) {
      label %cont: {
        %EnvModify($context, "i",
          func(x1, x2) { %PrimMult(x1, x2) },
          func() {
            %EnvPrepostOp($context, "x",
              func(x1, x2) { x1 - x2) }, false, $strict)
          }, $strict)
      };;
      #while_loop()
    } else { empty }
  }) #while_loop()
};;
%EnvGet($context, "i", $strict)
```

Context manipulation

The JavaScript code

```
var x=5, i=1
```

generates two blocks of λ_{JS} code:

- Environment initialization (at the beginning):

```
%EnvDefineVar($vcontext, "x", false, $strict);
%EnvDefineVar($vcontext, "i", false, $strict);
```

- Value assignment (at initialization site):

```
%EnvAssign($context, "x", func() {5.}, $strict);
%EnvAssign($context, "i", func() {1.}, $strict); empty
```

JS contexts represented as objects

The two semicolons

- The single semicolon `;` is a standard semicolon:
  - Simple semantics: evaluate lhs, ignore the result, evaluate rhs, return its result
  - Exceptions and breaks “pass through”, no special behavior
  - `v; e` equivalent to just `e`
- The double semicolon `;;` treats `empty` specially
  - Our solution for handling ES5 statement results correctly (was not in original λ_{JS})
  - If rhs returns `empty`, it is overwritten by the lhs value
  - This holds also for breaks: `v; break lbl empty` equivalent to `break lbl v`

The while loops

A while loop `while(cond){ body }` is desugared to:

```
label %brk: {
  rec (#while_loop = func() {
    if (%ToBoolean(cond)) {
      label %cont: body;;
      #while_loop()
    } else { empty }
  }) #while_loop()
}
```

- The labels enable handling of JS `continue` and `break` statements in the body of the loop
- The double semicolon handles the result value for the loop
  - Interesting inconsistency involving `break` found in ES5; we managed to get this fixed in ES6: `https://esdiscuss.org/topic/loop-unrolling-and-completion-values-in-es6`

Environment operations

Example – a compound assignment `x *= rhs` is desugared to:

```
%EnvModify($context, "x",
  func(x1, x2) { %PrimMult(x1, x2) },
  func() { rhs }, $strict)
```

Primitive multiplication `PrimMult` is defined (basically) as:

```
func(x1, x2) { %ToNumber(x1) * %ToNumber(x2) }
```

- `func` used to delay rhs evaluation until lhs is evaluated
- In original λ_{JS}, `lhs *= rhs` was desugared the same as `lhs = lhs * rhs` – gives different results if `lhs` has side effects! (e.g. getters/setters)

3

Problems with λ_{JS}

Original λ_{JS} is buggy

- By “buggy”, I mean not consistent with ES5 spec
- Over 30 issues found, including some serious ones:
  `https://github.com/tilk/LambdaCert/wiki/Issues-with-LambdaS5`
- Numerous changes were needed, in the core language and the desugaring

Object field access

- Numerous issues, involving getters/setters and exotic objects
- The most shocking one:

```
var x = {set y(v) { return 1 }};
x.y = 2 // returns 1 instead of 2
```

- We moved field getting/setting out of core semantics into the desugaring
  - Pros: simpler core semantics, easier to make correct, easy handling of exotic objects
  - Cons: more complexity in the desugaring

Exotic objects

- The original λ_{JS} handled exotic objects by special-casing
- This leads to subtle bugs
  - `length` property of arrays was handled in field assignment, but not in `defineProperty`:

```
x = [1,2,3];
Object.defineProperty(x, "length", {value: 1});
x[2] // returns 3 instead of undefined
```

  - `instanceof` was not special-cased for `bind` objects:

```
function f(){};
var o = new f(), fb = f.bind();
o instanceof fb // throws instead of returning true
```

- Our solution: object internal methods as internal fields, definitions close to the spec

4

A tale of one discrepancy

JavaScript statement result values

- Javascript statements can return values
- But they can be only observed by `eval` (or REPL)
- Some statements do not return a value (e.g. empty blocks)
- Return value of a complex statement is the last value returned by a sub-statement

Examples:

- `{}` does not return a value
- `x=1` returns 1
- and so does `{}; x=1` and `x=1; {}`

While loop in LambdaJS

A while loop is translated (simplified) to:

```
rec (loop = func() {
  if (condition) {
    body;; loop()
  } else {
    empty
  }
}) { loop() }
```

The LambdaJS double semicolon operator includes empty result handling.

Funny example

Take a look at this snippet:

```
var c; a: { c=1; while(true){ if (c) c=0; else break a; } }
```

The V8 and SpiderMonkey return 0, and so does LambdaJS.
But according to ES5, the result is 1:

12.6.2 The `while` Statement

The production *IterationStatement* : **`while`** **`(`** *Expression* **`)`** *Statement* is evaluated as follows:

1. Let V = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, V, empty).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let V = *stmt*.value.
   e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is break and *stmt*.target is in the current label set, then
         1. Return (normal, V, empty).
      ii. If *stmt* is an abrupt completion, return *stmt*.

Let's unroll

We unroll one iteration:

```
var c; a: { c=1; if (true) { if (c) c=0; else break a;
  while(true){ if (c) c=0; else break a; } } }
```

And now ES5 agrees with the implementations that the result is 0.

Let's unroll

We unroll one iteration:

```
var c; a: { c=1; if (true) { if (c) c=0; else break a;
  while(true){ if (c) c=0; else break a; } } }
```

And now ES5 agrees with the implementations that the result is 0.

We reported this to the EcmaScript committee, and the specification issue is now fixed!

`esdiscuss.org/topic/loop-unrolling-and-completion-values-in-es6`

5

Proof architecture

General proof architecture

- JSCert and λ_{JS} heaps related by bisimulation
- Proof by induction on the depth of the pretty-big-step λ_{JS} semantics derivation
- Induction hypotheses for statements, expressions and function calls
- Tactics for forward reasoning, automated (like: if I see an assumption of the form Φ, derive Ψ)
- Final conclusion (JSCert semantics derivation, relationship of results) proved by `eauto` with custom hint database

Maintaining invariants

- At all times during evaluation, invariants must hold which say that S5 and JSCert state is correctly related
- Split into environment and heap invariants:

```
Record state_invariant
      : fact_set → J.state → L.store → Prop.
Record context_invariant
      : fact_set → J.execution_ctx → L.ctx → Prop.
```

- The context invariant uses the “fact set” (includes heap bisimulation) only positively, therefore is monotonous on the “fact set” – simplifies proofs, because facts are never removed

Context invariant

Enforces that:

- The JSCert evaluation context is correctly represented in S5 (the *this* binding, the *strict* flag, the stack of environment records)
- The S5 environment definitions are accessible (and not shadowed) in the S5 context
- Preallocated JavaScript objects are correctly related to preallocated S5 objects (named in the S5 environment)
- The Javascript global environment is correctly related to S5

State invariant

Enforces that:

- The bisimulation relation is a bijection
- Every JavaScript object (or environment record) is related to some S5 object
- Related pairs of JavaScript and S5 objects satisfy necessary properties
- S5 helper objects have a correct form

The soundness theorem for statements

```
Definition th_stat k jt := forall BR jst jc c st st' r,
  context_invariant BR jc c →
  state_invariant BR jst st →
  L.red_exprh k c st (js_stat_to_ljs jt) (L.out_ter st' r) →
  exists BR' jst' jr,
  J.red_stat jst jc jt (J.out_ter jst' jr) ∧
  state_invariant BR' jst' st' ∧
  BR \c BR' ∧
  res_related BR' jst' st' jr r.
```

If...

- invariant holds for the initial state
- desugared program terminates in S5

Then...

- original program terminates in JSCert
- invariant holds for the final state
- the new fact set extends the initial one
- results are related

The problem with expressions

- expressions in ES5 (and thus in JSCert) sometimes return references
- solution: relate the values that would be produced if *GetValue* was called on the reference

```
Definition th_expr k je := forall BR jst jc c st st' r,
  context_invariant BR jc c →
  state_invariant BR jst st →
  L.red_exprh k c st (js_expr_to_ljs je) (L.out_ter st' r) →
  exists BR' jst' sr,
  J.red_spec jst jc (J.spec_expr_get_value je) sr ∧
  ((exists jv v, sr = J.specret_val jst' jv ∧
      r = L.res_value v ∧ value_related BR' jv v) ∨
   (exists jr, sr = J.specret_out jr ∧ J.abort jr ∧
      J.res_type jr = J.restype_throw ∧
      res_related BR' jst' st' jr r))
  state_invariant BR' jst' st' ∧
  BR \c BR' ∧
  res_related BR' jst' st' jr r.
```

Speeding up (e)auto

- Unfolding negation **not a good idea** – *auto* does goal-directed, depth-first search, not a good choice for proving contradictions
- Heurestic tactics for negation + NNF transformation via *rew_logic* much better
- Some hints (including those from TLC) were leading to infinite or unnecessary branches in proof search
- My solution: *nocore*, minimal hint database

Fast inversions

- The standard *inversion* tactic in Coq works very slowly on large inductive predicates (*red_expr* has ~100 constructors)
- And it generates large proof terms; this makes Coq use incredible amounts of memory
- Simple solution – precompute inversion principles using *Derive Inversion* and write a custom inversion predicate with pattern-matching
- I generate the boilerplate Coq code (~75 inversion principles + pattern matching) using a script

The need for fast lookups

- The S5 environment file is converted into an ~800k long file, with a list of ~400 definitions:

```
Definition ctx_items : list (string * value)
```

- In proofs, it is often needed to find a definition in this list and generate a membership certificate in form of a `Mem` value
- `repeat (eapply Mem_here || eapply Mem_next)`
  too slow (takes a few seconds)
- Solution – instead of constructing a proof, compute:

```
Lemma fast_string_assoc_mem : forall A k l (v : A),
  fast_string_assoc k l = Some v → Mem (k, v) l.
```

- Avoid eager unfolding of definitions by `cbv` by adding `-[ids...]`

Thank you for your attention!