LambdaCert

S5 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 S5

- 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 S5

- 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/`

S5 has bugs

S5 is tested (test262), but I found some bugs in it:

- variable binding (in *try-catch*, *getters*/*setters*)
- recursive function expressions
- *continue* with a label
- *reduce* and *reduceRight* for arrays

Possibly more to be found...

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

2

Technical details

S5 core at a glance

Available datatypes:

- Literals (booleans, numbers, null, undefined, empty)
- Functions (basically lambdas with multiple arguments)
- JS-like objects (model: data fields, getters/setters, prototypes, code, security properties)

Features in the language:

- No loops etc., only letrec
- No assignment operator (but objects mutable)
- Conditionals: if expressions only
- Control flow: exceptions, label/throw
- eval for running Javascript code

Formalized S5 core

- Pretty-big-step style

```
Inductive red_expr : ctx → store → ext_expr → out → Prop
```

- `ctx` – maps variables to values
- `store` – maps object pointers to objects
- $\approx$ 100 rules (JSCert has $\approx$ 800)
- $\approx$ 500 lines (JSCert has $\approx$ 4500)

S5 interpreter

- Least-fixed point outside the main definition, as in JSRef

```
Definition runs_type := ctx → store → expr → result.
Definition eval : runs_type → runs_type.
Definition runs : nat → runs_type. (* iterated eval *)
```

- `result` type includes *bottom* and readable error messages

Equivalence proof

- Soundness:

```
Lemma runs_correct : forall k c st e o,
  runs k c st e = result_some o →
  red_expr c st e o.
```

- Completeness:

```
Lemma runs_complete : forall c st e o,
  red_expr c st e o →
  exists k, runs k c st e = result_some o.
```

- Height-indexed version of `red_expr` used for proving completeness (TLC's `induct_height` used)

```
Lemma runs_complete_lemma : forall k c st e o,
    red_exprh k c st e o → runs k c st e = result_some o.
```

Desugaring function

- Split into two parts:

```
(* JS to ExprJS *)
Fixpoint js_expr_to_ejs (e : J.expr) : E.expr := ...
with js_stat_to_ejs (t : J.stat) : E.expr := ...
with js_prog_to_ejs (p : J.prog) : E.prog := ...
...
(* ExprJS to S5 *)
Fixpoint ejs_to_ljs (e : E.expr) : L.expr := ...
```

- ExprJS is basically simplified Javascript:
  - no statements, only expressions
  - no *continue*, no *return*, only basic *label* and *break*
  - no *for*, only *while*
  - no else-less *if*

Proving soundness of S5 wrt JSCert

Issues to solve:

- Different representations of heaps, data and contexts
- References all over the place in ES5, no references in S5

My current approach:

- Heap bisimulation

```
Definition object_bisim :=
  J.object_loc → L.object_ptr → Prop.
```

- Maintaining invariants

```
Inductive state_invariant : object_bisim →
  J.state → J.execution_ctx → L.ctx → L.store → Prop
```

- Predicates for relating S5 and JSCert

Proving soundness of S5 wrt JSCert

Relating relevant parts of S5 and JSCert semantics:

```
Inductive value_related :
  object_bisim → J.value → L.value → Prop
Inductive object_related :
  object_bisim → J.object → L.object → Prop
Inductive resvalue_related :
  object_bisim → J.resvalue → L.value → Prop
Inductive res_related :
  object_bisim → J.state → L.store → J.res → L.res → Prop
```

All objects related by the bisimulation relation must also be related by `object_related`.

The soundness theorem for statements

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

If...

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

Then...

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

The problem with expressions

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

Parts already proven

Statements:

- Expression statement
- Blocks
- Conditionals
- Break, continue, return, label
- Throw

Expressions:

- Literals
- Conditionals