A Principled Intermediate Language for JavaScript Verification

Daiva Naudžiūnienė

Imperial College London

Resource Reasoning Meeting

The Team

Philippa Gardner

Gareth Smith

Thomas Wood

José Fragoso Santos

Petar Maksimović

Running Example: Where is Alice?

```
var Person = function (x) {
  this.name = x;
}

Person.prototype.sayHi = function () {
  return "Hi! I am " + this.name;
}

var alice = new Person("Alice");
alice.sayHi();
```

Where is Alice?

```
var Person = function (x) {
   this.name = x;
}
Person.prototype.sayHi = function() {
   return "Hi! I am " + this.name;
}
var alice = new Person("Alice");
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person

2. Evaluate the function literal,

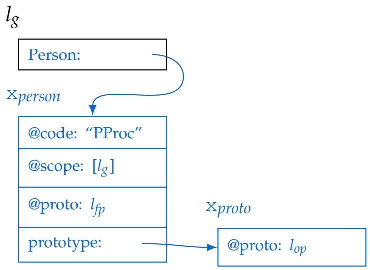
and assign to "Person.prototype.sayHi"

3. Create new object

4. Evaluate function body with *this* being an object created in Step 3

5. Assign new object to var alice

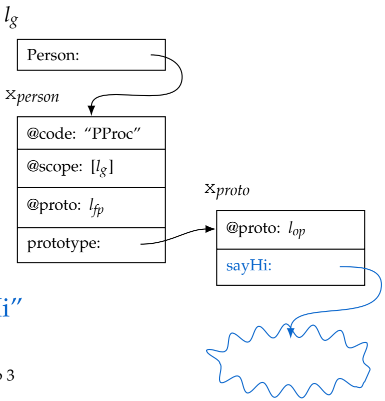
6. Call alice.sayHi()



Where is Alice?

```
var Person = function (x) {
   this.name = x;
}
Person.prototype.sayHi = function() {
   return "Hi! I am " + this.name;
}
var alice = new Person("Alice");
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,
   and assign to "Person.prototype.sayHi"
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice
6. Call alice.sayHi()



Where is Alice?

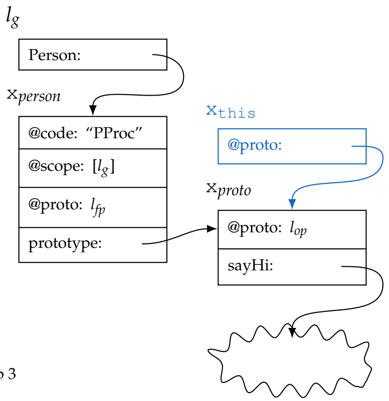
```
var Person = function (x) {
   this.name = x;
}
Person.prototype.sayHi = function() {
   return "Hi! I am " + this.name;
}
var alice = new Person("Alice");
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,

   and assign to "Person.prototype.sayHi"

3.Create new object

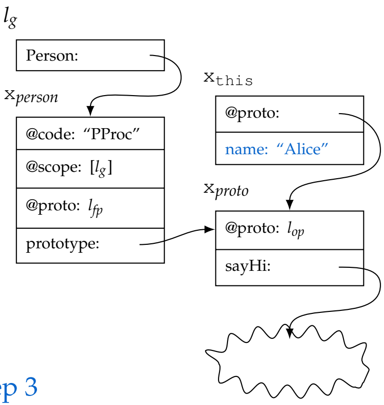
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice
6. Call alice.sayHi()



Where is Alice?

```
var Person = function (x) {
  this.name = x;
}
Person.prototype.sayHi = function() {
  return "Hi! I am " + this.name;
}
var alice = new Person("Alice");
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,
   and assign to "Person.prototype.sayHi"
3. Create new object
4. Evaluate function body
   with *this* being an object created in Step 3
5. Assign new object to var alice
6. Call alice.sayHi()

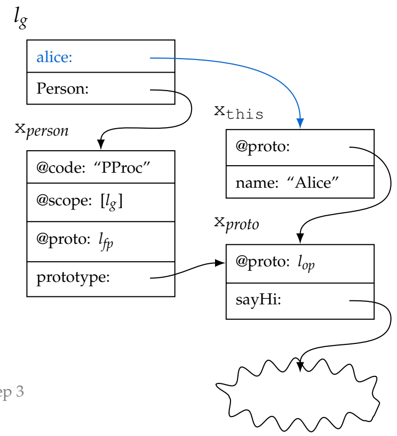


Where is Alice?

```
var Person = function (x) {
   this.name = x;
}
Person.prototype.sayHi = function() {
   return “Hi! I am ” + this.name;
}
var alice = new Person(”Alice”);
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,

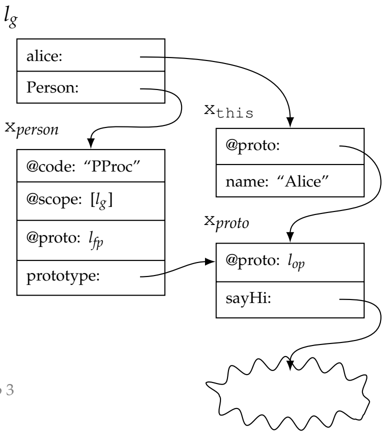
   and assign to “Person.prototype.sayHi”
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice
6. Call alice.sayHi()



Where is Alice?

```
var Person = function (x) {
   this.name = x;
}
Person.prototype.sayHi = function() {
   return “Hi! I am ” + this.name;
}
var alice = new Person(”Alice”);
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,
   and assign to “Person.prototype.sayHi”
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice
6. Call alice.sayHi()



Running Wrong Example: We Cannot Find Alice

```
var Person = function (x) {
  this.name = x;
}

Person.prototype.sayHi = function () {
  return "Hi! I am " + this.name;
}

var alice = Person("Alice");
alice.sayHi();
```

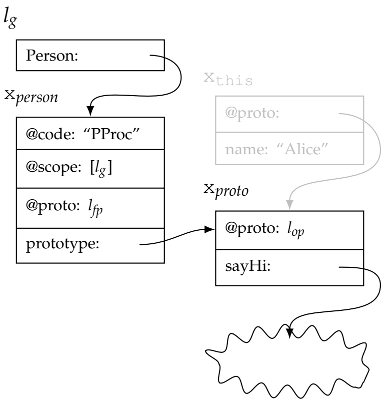
We cannot find Alice

```
var Person = function (x) {
    this.name = x;
}
Person.prototype.sayHi = function() {
  return "Hi! I am " + this.name;
}
var alice = Person("Alice");
alice.sayHi();
```

1. Evaluate the function literal and assign to var Person
2. Evaluate the function literal,

   and assign to "Person.prototype.sayHi"
3. ~~Create new object~~
4. Evaluate function body

   with *this* being undefined
5. Assign return value to var alice
6. Call alice.sayHi()

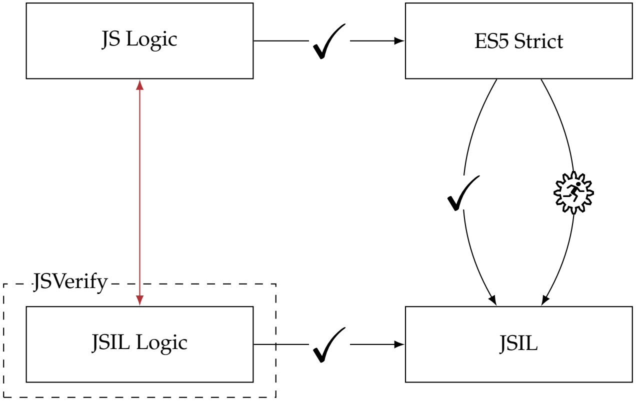


Verifying JavaScript Programs

‘Towards a Program Logic for JavaScript’, POPL’12, Gardner, Maffeis, and Smith.

Overall Project

JSVerify: A verification tool for JavaScript programs based on an intermediate language **JSIL**

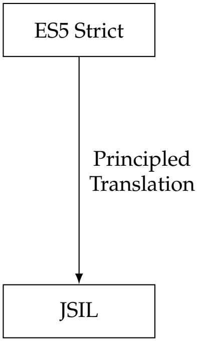


From ES5 Strict to JSIL - Our choices

We implemented, tested, and proved correct a principled translation from ES5 Strict to JSIL.

JSIL was specifically designed as a simple verification language for Javascript:

- small language;
- simple semantics;
- similar memory model to JavaScript's memory model.



JSIL

Simple goto language:

$$\begin{array}{lcl} \mathsf{C} \in \mathtt{Cmd} & \triangleq & \mathtt{x} := \mathtt{E} \\ & & \mid \mathtt{skip} \mid \mathtt{goto}\ \mathtt{l} \mid \mathtt{goto}\ [\mathtt{E}]\ \mathtt{l}_1,\ \mathtt{l}_2 \end{array}$$

`l` denotes a label and `E` is an expression with no side effects.

JSIL

Procedure calls:

$$
\begin{aligned}
C \in \texttt{Cmd} \quad &\triangleq \quad \texttt{x} := \texttt{E} \\
&\quad \mid \texttt{skip} \mid \texttt{goto}\ \texttt{l} \mid \texttt{goto}\ [\texttt{E}]\ \texttt{l}_1,\ \texttt{l}_2 \\
&\quad \boxed{\mid \texttt{x}\ :=\ \texttt{p}(\texttt{E}, \ldots, \texttt{E})}
\end{aligned}
$$

$\texttt{p} \in \{\texttt{E}, \texttt{eval}, \texttt{built-in-PId}\}$

JSIL

JavaScript heap commands:

$$\begin{array}{lcl}
C \in \texttt{Cmd} & \triangleq & \texttt{x} := \texttt{E} \\
 & & \mid \texttt{skip} \mid \texttt{goto l} \mid \texttt{goto} \, [\texttt{E}] \, \texttt{l}_1, \, \texttt{l}_2 \\
 & & \mid \texttt{x} := \texttt{p}(\texttt{E}, \ldots, \texttt{E}) \\
 & & \boxed{\begin{array}{l}
 \mid \texttt{x} := \texttt{new}() \mid \texttt{x} := \texttt{hasField}(\texttt{E}, \, \texttt{E}) \\
 \mid \texttt{x} := [\texttt{E}, \, \texttt{E}] \mid [\texttt{E}, \, \texttt{E}] := \texttt{E} \\
 \mid \texttt{x} := \texttt{delete}(\texttt{E}, \, \texttt{E})
 \end{array}}
\end{array}$$

JSIL

Prototype-based inheritance:

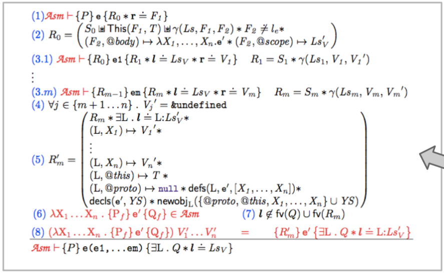
$$\begin{array}{lcl}
C \in \texttt{Cmd} & \triangleq & \texttt{x := E} \\
 & & \mid \texttt{skip} \mid \texttt{goto l} \mid \texttt{goto [E] l}_1\texttt{, l}_2 \\
 & & \mid \texttt{x := p(E,...,E)} \\
 & & \mid \texttt{x := new()} \mid \texttt{x := hasField(E, E)} \\
 & & \mid \texttt{x := [E, E]} \mid \texttt{[E, E] := E} \\
 & & \mid \texttt{x := delete(E, E)} \\
 & & \boxed{\begin{array}{l} \mid \texttt{x := protoField(E, E)} \\ \mid \texttt{x := protoObj(E, E)} \end{array}}
\end{array}$$

JSIL

$$
\begin{array}{rcl}
C \in \texttt{Cmd} & \triangleq & \texttt{x} := \texttt{E} \\
& & \mid \texttt{skip} \mid \texttt{goto l} \mid \texttt{goto}\,[\texttt{E}]\;\texttt{l}_1,\;\texttt{l}_2 \\
& & \mid \texttt{x} \;:=\; \texttt{p}(\texttt{E},\ldots,\texttt{E}) \\
& & \mid \texttt{x} \;:=\; \texttt{new}() \mid \texttt{x} \;:=\; \texttt{hasField}(\texttt{E},\,\texttt{E}) \\
& & \mid \texttt{x} \;:=\; [\texttt{E},\,\texttt{E}] \mid [\texttt{E},\,\texttt{E}] \;:=\; \texttt{E} \\
& & \mid \texttt{x} \;:=\; \texttt{delete}(\texttt{E},\,\texttt{E}) \\
& & \mid \texttt{x} \;:=\; \texttt{protoField}(\texttt{E},\,\texttt{E}) \\
& & \mid \texttt{x} \;:=\; \texttt{protoObj}(\texttt{E},\,\texttt{E})
\end{array}
$$

$$
\boxed{
\begin{array}{rcl}
\texttt{Procedure} & \triangleq & \texttt{proc}\;\texttt{PId}(\texttt{x}_1,\ldots,\texttt{x}_n)\{ \\
& & \quad \texttt{0}: \texttt{C}_0 \\
& & \quad \texttt{1}: \texttt{C}_1 \\
& & \quad \ldots \\
& & \quad \texttt{m}: \texttt{C}_\texttt{m} \\
& & \}
\end{array}
}
$$

JSIL Logic versus JS Logic



$$
\begin{array}{l}
(1)\, \mathcal{A}sm \vdash \{P\}\, \mathsf{e}\, \{R_0 * \mathbf{r} \doteq F_1\} \\
(2)\ R_0 = \begin{pmatrix} S_0 \boxast \mathsf{This}(F_1, T) \boxast \gamma(Ls, F_1, F_2) * F_2 \not\doteq l_e * \\ (F_2, @body) \mapsto \lambda X_1, \ldots, X_n.\mathsf{e}' * (F_2, @scope) \mapsto Ls'_V \end{pmatrix} \\
(3.1)\ \mathcal{A}sm \vdash \{R_0\}\, \mathtt{e1}\, \{R_1 * \boldsymbol{l} \doteq Ls_V * \mathbf{r} \doteq V_1\} \quad R_1 = S_1 * \gamma(Ls_1, V_1, V_1{}') \\
\vdots \\
(3.m)\ \mathcal{A}sm \vdash \{R_{m-1}\}\, \mathtt{em}\, \{R_m * \boldsymbol{l} \doteq Ls_V * \mathbf{r} \doteq V_m\} \quad R_m = S_m * \gamma(Ls_m, V_m, V_m{}') \\
(4)\ \forall j \in \{m+1 \ldots n\}\ .\ V_j{}' = \mathtt{\&undefined} \\
(5)\ R'_m = \begin{pmatrix} R_m * \exists \mathrm{L}\ .\ \boldsymbol{l} \doteq \mathrm{L}{:}Ls'_V * \\ (\mathrm{L}, X_1) \mapsto V_1{}' * \\ \vdots \\ (\mathrm{L}, X_n) \mapsto V_n{}' * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \mathtt{null} * \mathsf{defs}(\mathrm{L}, \mathsf{e}', [X_1, \ldots, X_n]) * \\ \mathsf{decls}(\mathsf{e}', YS) * \mathsf{newobj}_{\mathrm{L}}(\{@proto, @this, X_1, \ldots, X_n\} \cup YS) \end{pmatrix} \\
(6)\ \lambda \mathrm{X}_1 \ldots \mathrm{X}_n\ .\ \{\mathrm{P}_f\}\, \mathsf{e}'\, \{\mathrm{Q}_f\} \in \mathcal{A}sm \qquad (7)\ \boldsymbol{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_m) \\
(8)\ (\lambda \mathrm{X}_1 \ldots \mathrm{X}_n\ .\ \{\mathrm{P}_f\}\, \mathsf{e}'\, \{\mathrm{Q}_f\})\ V'_1 \ldots V'_n \quad = \quad \{R'_m\}\, \mathsf{e}'\, \{\exists \mathrm{L}\ .\ Q * \boldsymbol{l} \doteq \mathrm{L}{:}Ls'_V\} \\
\hline
\mathcal{A}sm \vdash \{P\}\, \mathsf{e}(\mathtt{e1}, \ldots \mathtt{em})\, \{\exists \mathrm{L}\ .\ Q * \boldsymbol{l} \doteq Ls_V\}
\end{array}
$$

JavaScript

Logic Rules of Function / Procedure Call

JSIL

$$
\frac{\mathtt{S}(\mathtt{fid}) = \lambda \mathtt{x}_1, \ldots, \mathtt{x}_n.\, \{P\}\, m\, \{Q * \mathtt{ret} \doteq \mathtt{E}\} \quad \forall_{k<j\leq n}\, \mathtt{E}_j = \text{undefined}}{\mathtt{S} \vdash \{P[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n]\}\ \mathtt{x} := \mathtt{fid}(\mathtt{E}_1, \ldots, \mathtt{E}_k)\ \{Q[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n] * \mathtt{x} \doteq \mathtt{E}[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n]\}}
$$

However...

The complexity of JavaScript does not disappear. It has moved to the code generated by the translation

Back to Example: Where is Alice?

```
var Person = function (x) {
    this.name = x;
}

var alice  = new Person("Alice");
```

Translating: Where is Alice in JSIL?

- The translation generates a top level procedure for each function literal.
- The translation generates a special procedure `main` for the global code.
- No nesting of procedures.

JavaScript Code	**JSIL Code**
var Person = **function** (x) { ... }	`proc PProc`($\mathtt{x}_{sc}$, $\mathtt{x}_{this}$, `x`){...}
Whole Program	`proc main`(){...}

Translating: Where is Alice in JSIL?

JavaScript Code

```
var Person = function (x) {
    this.name = x;
}
var alice = new Person("Alice");
```

1. Evaluate the function literal

2. Assign function object to var Person

3. Create new object

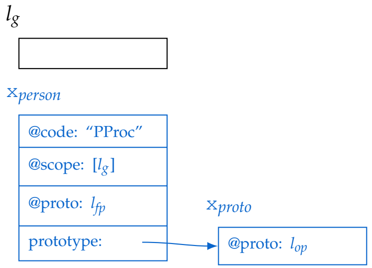
4. Evaluate function body with

this being an object created in Step 3

5. Assign new object to var alice

JSIL Code

$x_{proto} := \texttt{new}()$
$[x_{proto}, @proto] := l_{op}$
$x_{person} := \texttt{new}()$
$[x_{person}, @code] := \text{“PProc”}$
$[x_{person}, @scope] := [l_g]$
$[x_{person}, @proto] := l_{fp}$
$[x_{person}, \text{“prototype”}] := x_{proto}$



Translating: Where is Alice in JSIL?

JavaScript Code

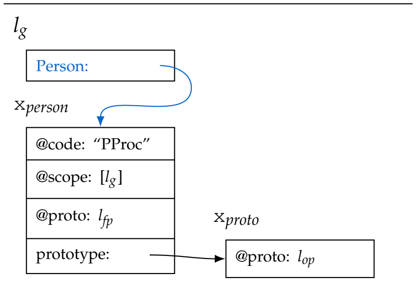
```
var Person = function (x) {
    this.name = x;
}
var alice = new Person("Alice");
```

1. Evaluate the function literal
2. Assign function object to var Person
3. Create new object
4. Evaluate function body with

   this being an object created in Step 3
5. Assign new object to var Alice

JSIL Code

$[l_g, \text{“}\mathit{Person}\text{”}] := \mathrm{x}_{\mathit{person}}$



Translating: Where is Alice in JSIL?

JavaScript Code

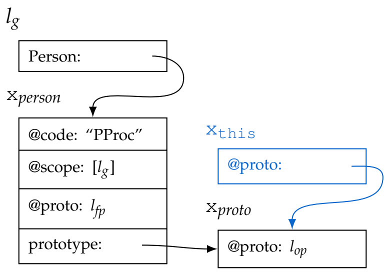
```
var Person = function (x) {
  this.name = x;
}
var alice = new Person("Alice");
```

1. Evaluate the function literal
2. Assign function object to var Person
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice

JSIL Code

$\mathtt{x}_{this} := \mathtt{new}()$

$[\mathtt{x}_{this}, @proto] := \mathtt{x}_{proto}$



Translating: Where is Alice in JSIL?

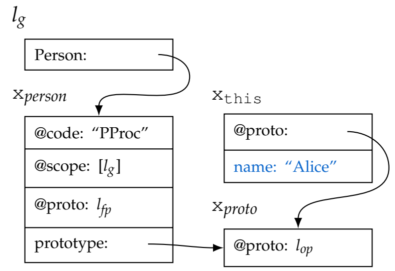
JavaScript Code

```
var Person = function (x) {
   this.name = x;
}
var alice = new Person("Alice");
```

1. Evaluate the function literal
2. Assign function object to var Person
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice

JSIL Code

$\mathtt{x}_{sc} := [\mathtt{x}_{person}, @scope]$
$\mathtt{x}_{f} := [\mathtt{x}_{person}, @code]$
$\mathtt{x}_{ret} := \mathtt{x}_{f}(\mathtt{x}_{sc}, \mathtt{x}_{this}, \text{“}Alice\text{”})$



Translating: Where is Alice in JSIL?

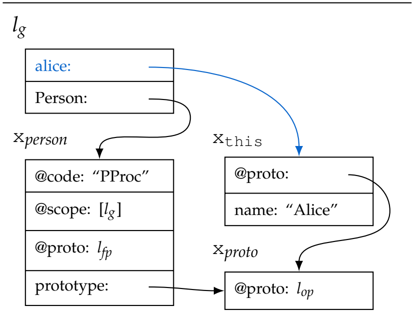
JavaScript Code

```
var Person = function (x) {
    this.name = x;
}
var alice = new Person("Alice");
```

1. Evaluate the function literal
2. Assign function object to var Person
3. Create new object
4. Evaluate function body with *this* being an object created in Step 3
5. Assign new object to var alice

JSIL Code

$[l_g, \text{"alice"}] := \mathtt{x}_{this}$



Back to Wrong Example: We Cannot Find Alice

```
var Person = function (name) {
    this .name = name;
}

var alice  = Person("Alice");
```

We “forgot” the new.

Translating: We Cannot Find Alice

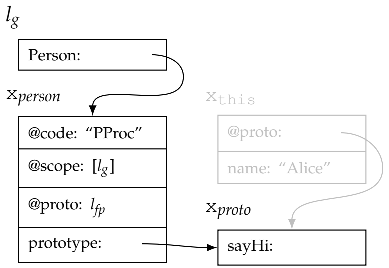
JavaScript Code

```
var Person = function (x) {
    this.name = x;
}
var alice = Person("Alice");
```

1. Evaluate the function literal
2. Assign function object to var Person
3. ~~Create new object~~
4. Evaluate function body with *this* being undefined
5. Assign return value to var alice

JSIL Code

$x_{sc} := [x_{person}, @scope]$
$x_f := [x_{person}, @code]$
$x_{ret} := x_f(x_{sc}, \texttt{undefined}, \text{“}Alice\text{”})$



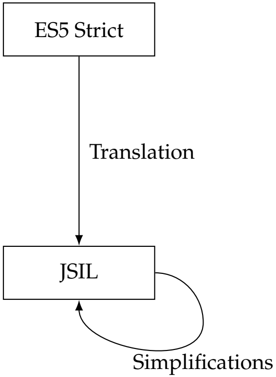
Simplifications

Naive translation, but makes soundness proof simple!

$$\begin{tikzcd}\boxed{\text{ES5 Strict}} \arrow[d, "\text{Translation}"] \\ \boxed{\text{JSIL}}\end{tikzcd}$$

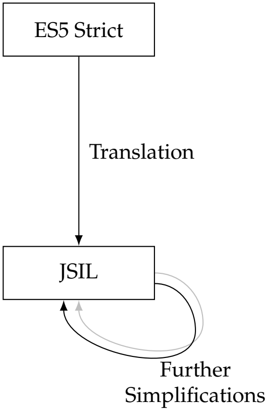
Simplifications

Standard compiler optimizations, e.g. constant propagation, dead code elimination, algebraic simplifications etc.

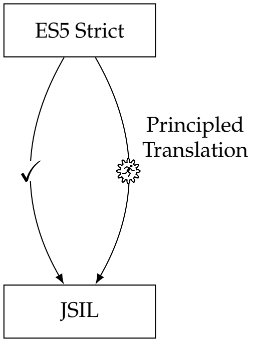


Simplifications

Further simplifications using symbolic execution.



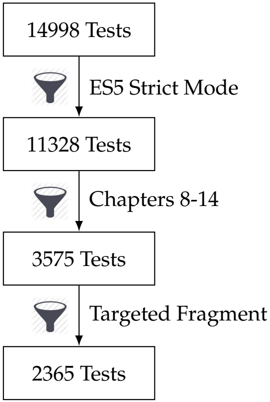
Trusted Translation



- Proven correct with respect to an operational semantics
- Tested using Test262

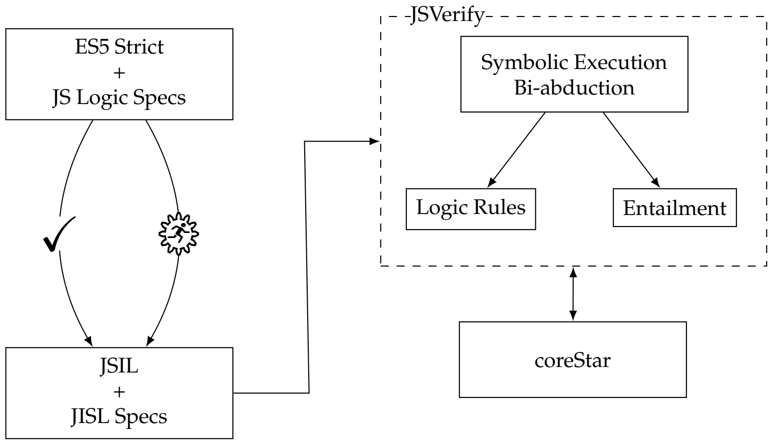
Validating the JSIL Compiler

- We have used ES6 *Test262*
- Targeted ES5 Strict Subset Chapters 8-14 except for:
  - Getters and Setters
  - Arguments Object
  - Property attributes
- We pass 100% of our targeted ES5 Strict subset



JSVerify

- Program logic for JSIL and a symbolic execution tool based on Separation Logic



JSVerify - Specifying and Verifying the Example

JavaScript Code

```
var Person = function (x) {
    this.name = x
}
var alice = new Person("Alice");
```

JSIL Code

$\{(\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \_\}$

$\mathtt{PProc}(\mathtt{x}_{sc}, \mathtt{x}_{this}, \mathtt{x})\{\ldots\}$

$\left\{ \begin{array}{l} (\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \mathtt{x}* \\ \mathtt{ret} \doteq \mathtt{undefined} \end{array} \right\}$

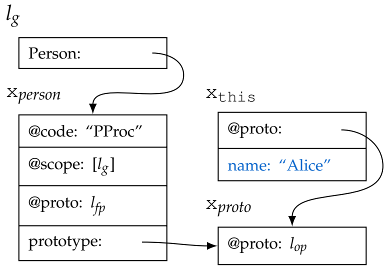
Specifying and Verifying the Example

Correct Example

$$\left\{\begin{array}{l}(\mathtt{x}_{person}, @scope) \mapsto [l_g] * \\ (\mathtt{x}_{person}, @code) \mapsto \text{“}PProc\text{”} * \\ (\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \varnothing\end{array}\right\}$$

$$\begin{array}{l}\mathtt{x}_{sc} := [\mathtt{x}_{person}, @scope] \\ \mathtt{x}_f := [\mathtt{x}_{person}, @code] \\ \mathtt{x}_{ret} := \mathtt{x}_f(\mathtt{x}_{sc}, \mathtt{x}_{this}, \text{“}Alice\text{”})\end{array}$$

$$\left\{\begin{array}{l}(\mathtt{x}_{person}, @scope) \mapsto [l_g] * \\ (\mathtt{x}_{person}, @code) \mapsto \text{“}PProc\text{”} * \\ (\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \text{“}Alice\text{”} * \\ \mathtt{x}_{sc} \doteq [l_g] * \mathtt{x}_f \doteq \text{“}PProc\text{”} * \\ \mathtt{x}_{ret} \doteq \mathtt{undefined}\end{array}\right\}$$



The spec:

$$\begin{array}{c}\{(\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \_\} \\ \mathtt{PProc} \\ \{(\mathtt{x}_{this}, \text{“}name\text{”}) \mapsto \mathtt{x} * \mathtt{ret} \doteq \mathtt{undefined}\}\end{array}$$

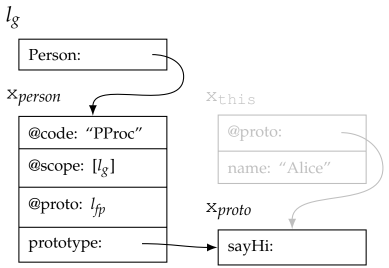
Specifying and Verifying the Wrong Example

Wrong Example

$$\left\{ \begin{array}{l} (\mathtt{x}_{person}, \mathit{@scope}) \mapsto [l_g] * \\ (\mathtt{x}_{person}, \mathit{@code}) \mapsto \text{“}\mathit{PProc}\text{”} \end{array} \right\}$$

$$\begin{array}{l} \mathtt{x}_{sc} := [\mathtt{x}_{person}, \mathit{@scope}] \\ \mathtt{x}_f := [\mathtt{x}_{person}, \mathit{@code}] \\ \mathtt{x}_{ret} := \mathtt{x}_f(\mathtt{x}_{sc}, \mathtt{undefined}, \text{“}\mathit{Alice}\text{”}) \end{array}$$

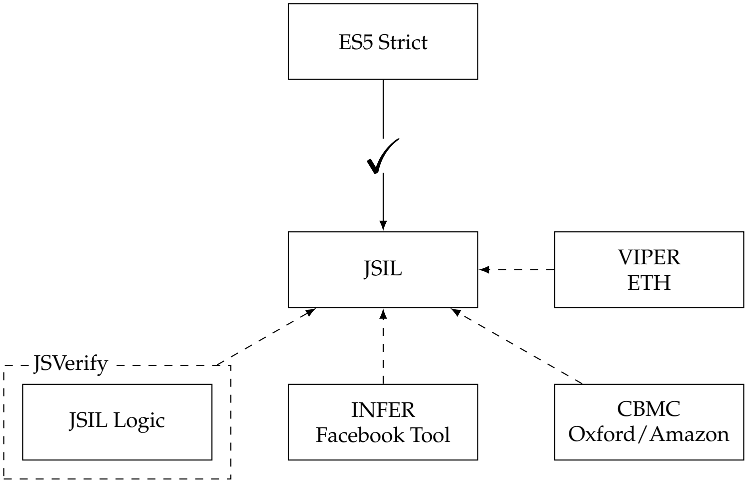
ERROR!



The spec:

$$\{(\mathtt{x}_{this}, \text{“}\mathit{name}\text{”}) \mapsto \_\}$$
$$\mathtt{PProc}$$
$$\{(\mathtt{x}_{this}, \text{“}\mathit{name}\text{”}) \mapsto \mathtt{x} * \mathtt{ret} \doteq \mathtt{undefined}\}$$

Connecting JSIL to other tools


ES5 Strict
✓
JSIL
VIPER
ETH
JSVerify
JSIL Logic
INFER
Facebook Tool
CBMC
Oxford/Amazon