An Abstract Machine for Asynchronous Programs with Closures and Priority Queues

Giorgio Delzanno
D. Ancona L. Franceschini M. Leotta
E. Prampolini M. Ribaudo F. Ricca
FUSTAQ Project[1]

DIBRIS, University of Genoa

[1] SEED 2016 project funded by DIBRIS, University of Genova

Overview

Plan

Motivatios

- Project on validation (testing, runtime verification, formal methods) of IoT applications developed in Node.js "Full Stack Quality of Javascript of Anything" funded by our University
- Node.js is a JavaScript runtime system built on Chrome's V8 JavaScript engine.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient
- Node.js is becoming a standard for IoT applications (for both server- and client-side software)

Motivatios

- Project on validation (testing, runtime verification, formal methods) of IoT applications developed in Node.js "Full Stack Quality of Javascript of Anything" funded by our University
- Node.js is a JavaScript runtime system built on Chrome's V8 JavaScript engine.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient
- Node.js is becoming a standard for IoT applications (for both server- and client-side software)

Motivatios

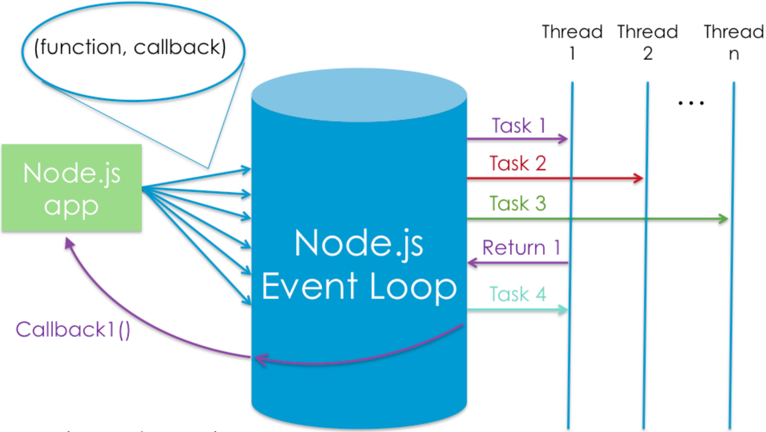
- Project on validation (testing, runtime verification, formal methods) of IoT applications developed in Node.js "Full Stack Quality of Javascript of Anything" funded by our University
- Node.js is a JavaScript runtime system built on Chrome's V8 JavaScript engine.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient
- Node.js is becoming a standard for IoT applications (for both server- and client-side software)

Motivatios

- Project on validation (testing, runtime verification, formal methods) of IoT applications developed in Node.js "Full Stack Quality of Javascript of Anything" funded by our University
- Node.js is a JavaScript runtime system built on Chrome's V8 JavaScript engine.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient
- Node.js is becoming a standard for IoT applications (for both server- and client-side software)

1 Node apps pass async tasks to the event loop, along with a callback

2 The event loop efficiently manages a thread pool and executes tasks efficiently...



3 ...and executes each callback as tasks complete

Node.js

var result = db.query("SELECT..."); // use result

VS

db.query("SELECT...", function (result) // use result);

Main Features of Node.js

- Built on top of Javascript
- Asynchronous calls to avoid synchronization primitive such as locks
- Priority queues to model different types of events (input/output, delayed calls, etc);

  Continuation-style programming: callbacks with highest priority
- Closures to handle variables used in callbacks but declared in outermost functions

Main Features of Node.js

- Built on top of Javascript
- Asynchronous calls to avoid synchronization primitive such as locks
- Priority queues to model different types of events (input/output, delayed calls, etc);

  Continuation-style programming: callbacks with highest priority
- Closures to handle variables used in callbacks but declared in outermost functions

Main Features of Node.js

- Built on top of Javascript
- Asynchronous calls to avoid synchronization primitive such as locks
- Priority queues to model different types of events (input/output, delayed calls, etc);

  Continuation-style programming: callbacks with highest priority
- Closures to handle variables used in callbacks but declared in outermost functions

Main Features of Node.js

- Built on top of Javascript
- Asynchronous calls to avoid synchronization primitive such as locks
- Priority queues to model different types of events (input/output, delayed calls, etc);

  Continuation-style programming: callbacks with highest priority
- Closures to handle variables used in callbacks but declared in outermost functions

Main Features of Node.js

- Built on top of Javascript
- Asynchronous calls to avoid synchronization primitive such as locks
- Priority queues to model different types of events (input/output, delayed calls, etc);

  Continuation-style programming: callbacks with highest priority
- Closures to handle variables used in callbacks but declared in outermost functions

Emitter

```
var EventEmitter = require('events');
var Emitter = new EventEmitter();
var msg = function msg() { console.log('ok'); }
Emitter.on('evt1', msg);
Emitter.emit('evt1');
while (true);
```

Emitter + Setimmediate

```
var EventEmitter = require('events');
var Emitter = new EventEmitter();
var msg = function msg() { console.log('ok'); }
Emitter.on('evt1', function () { setImmediate(msg); });
Emitter.emit('evt1');
while (true);
```

Closures and callbacks

```
function test(){
 var d = 5;
 var foo = function(){ d = 10; }
 process.nextTick(foo);
 setImmediate(() => { console.log(d) })
}
test();
```

Informal semantics

- `test` is called synchronously
- `foo` is delayed till the end of main (closure is stored in the heap)
- `console.log(d)` is postponed to the next tick (closure stored in the heap)
- when the main terminates `foo` updates `d`
- in the next loop tick `console.log` prints the updated value 10

Informal semantics

- `test` is called synchronously
- `foo` is delayed till the end of main (closure is stored in the heap)
- `console.log(d)` is postponed to the next tick (closure stored in the heap)
- when the main terminates `foo` updates `d`
- in the next loop tick `console.log` prints the updated value 10

Informal semantics

- `test` is called synchronously
- `foo` is delayed till the end of main (closure is stored in the heap)
- `console.log(d)` is postponed to the next tick (closure stored in the heap)
- when the main terminates `foo` updates `d`
- in the next loop tick `console.log` prints the updated value 10

Informal semantics

- `test` is called synchronously
- `foo` is delayed till the end of main (closure is stored in the heap)
- `console.log(d)` is postponed to the next tick (closure stored in the heap)
- when the main terminates `foo` updates `d`
- in the next loop tick `console.log` prints the updated value 10

Informal semantics

- `test` is called synchronously
- `foo` is delayed till the end of main (closure is stored in the heap)
- `console.log(d)` is postponed to the next tick (closure stored in the heap)
- when the main terminates `foo` updates `d`
- in the next loop tick `console.log` prints the updated value `10`

For Devs Only?

- Dev documentation is not clear at all
- Program semantics can be very hard to understand
- Non-determinism due to possible reorderings of events and delay of asynchronous operations
- Program transformations and design patterns are often used to simplify Node.js programs
- Formal semantics/reasoning to increase software quality!

For Devs Only?

- Dev documentation is not clear at all
- Program semantics can be very hard to understand
- Non-determinism due to possible reorderings of events and delay of asynchronous operations
- Program transformations and design patterns are often used to simplify Node.js programs
- Formal semantics/reasoning to increase software quality!

For Devs Only?

- Dev documentation is not clear at all
- Program semantics can be very hard to understand
- Non-determinism due to possible reorderings of events and delay of asynchronous operations
- Program transformations and design patterns are often used to simplify Node.js programs
- Formal semantics/reasoning to increase software quality!

For Devs Only?

- Dev documentation is not clear at all
- Program semantics can be very hard to understand
- Non-determinism due to possible reorderings of events and delay of asynchronous operations
- Program transformations and design patterns are often used to simplify Node.js programs
- Formal semantics/reasoning to increase software quality!

For Devs Only?

- Dev documentation is not clear at all
- Program semantics can be very hard to understand
- Non-determinism due to possible reorderings of events and delay of asynchronous operations
- Program transformations and design patterns are often used to simplify Node.js programs
- Formal semantics/reasoning to increase software quality!

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Our Proposal

- An Abstract Machine to describe the semantics of Asynchronous Programs with Priority Queues and Closures inspired to Node.js
- Built in two steps:
  - Host language with callback definitions and closures
  - Abstract machine (parametric on the pperational semantics of the host language) to describe event loop, continuations and callbacks with priorities
- Closures, the bridge between the two layers, are modeled via a shared heap
- Meta-interpreter built in Prolog to reason about all possible program executions (non determinism due to event triggering and termination of asynchronous operations)

Plan

Host Language

We introduce a host (imperative) language $\mathcal{L}$ defined as follows

- F is a set of function names.
- Var is a set of variables (it also contains function names in F)
- $Callback$ is the set of (anonymous) callback definitions of the form $\lambda\vec{x}.s$, where $\vec{x} \in Var^k$ are formal parameters and s is a list of statements
- Val contains primitive values and closures

Host Language

We introduce a host (imperative) language $\mathcal{L}$ defined as follows

- F is a set of function names.
- Var is a set of variables (it also contains function names in F)
- $Callback$ is the set of (anonymous) callback definitions of the form $\lambda\vec{x}.s$, where $\vec{x} \in Var^k$ are formal parameters and s is a list of statements
- Val contains primitive values and closures

Host Language

We introduce a host (imperative) language $\mathcal{L}$ defined as follows

- F is a set of function names.
- Var is a set of variables (it also contains function names in F)
- $Callback$ is the set of (anonymous) callback definitions of the form $\lambda\vec{x}.s$, where $\vec{x} \in Var^k$ are formal parameters and s is a list of statements
- Val contains primitive values and closures

Host Language

We introduce a host (imperative) language $\mathcal{L}$ defined as follows

- F is a set of function names.
- Var is a set of variables (it also contains function names in F)
- $Callback$ is the set of (anonymous) callback definitions of the form $\lambda\vec{x}.s$, where $\vec{x} \in Var^k$ are formal parameters and s is a list of statements
- Val contains primitive values and closures

Programs

Let B be a finite sequence of instructions in $Stmts^*$

- $let\ x = e\ in\ B$ where x is a local variable, e an expression denoting a primitive value,
- $let\ f_1 = \lambda\vec{y}_1.P_1, \ldots, f_k = \lambda\vec{y}_k.P_k\ in\ B$ where $P_1, \ldots, P_k$ are program expressions, they may contain *let* declarations to model nested callback declarations
- Example
  $P = let\ f = (let\ (cb = \lambda x.\ obs(x))\ in\ call(read, cb) \cdot f)\ in\ f()$

Programs

Let B be a finite sequence of instructions in $Stmts^*$

- *let* $x = e$ *in* B where x is a local variable, e an expression denoting a primitive value,
- *let* $f_1 = \lambda\vec{y}_1.P_1, \ldots, f_k = \lambda\vec{y}_k.P_k$ *in* B where $P_1, \ldots, P_k$ are program expressions, they may contain *let* declarations to model nested callback declarations
- Example
  $P = let\ f = (let\ (cb = \lambda x.\ obs(x))\ in\ call(read, cb) \cdot f)\ in\ f()$

Programs

Let B be a finite sequence of instructions in $Stmts^*$

- $let\ x = e\ in\ B$ where x is a local variable, e an expression denoting a primitive value,
- $let\ f_1 = \lambda\vec{y}_1.P_1, \ldots, f_k = \lambda\vec{y}_k.P_k\ in\ B$ where $P_1, \ldots, P_k$ are program expressions, they may contain *let* declarations to model nested callback declarations
- Example
  $P = let\ f = (let\ (cb = \lambda x.\ obs(x))\ in\ call(read, cb) \cdot f)\ in\ f()$

(Lightweight) Instruction Set

- $obs(e)$ to observe a certain event (a value)
- $store(x, e)$ to store a value (the evaluation of e) in the global or local variable x. We use the expression *any* to denote a value non deterministically selected from the set of values.
- $f(\vec{e})$ to synchronously invoke a callback f with the vector of parameters $\vec{e}$. Actual parameters are global or local variables.

(Lightweight) Instruction Set

- $obs(e)$ to observe a certain event (a value)
- $store(x, e)$ to store a value (the evaluation of e) in the global or local variable x. We use the expression *any* to denote a value non deterministically selected from the set of values.
- $f(\vec{e})$ to synchronously invoke a callback f with the vector of parameters $\vec{e}$. Actual parameters are global or local variables.

(Lightweight) Instruction Set

- $obs(e)$ to observe a certain event (a value)
- $store(x, e)$ to store a value (the evaluation of e) in the global or local variable x. We use the expression *any* to denote a value non deterministically selected from the set of values.
- $f(\vec{e})$ to synchronously invoke a callback f with the vector of parameters $\vec{e}$. Actual parameters are global or local variables.

Semantic Domains

- $Env = [Vars \rightarrow Loc]$
- $Closures = Env \times Callback$
- Val contains primitive values and closures
- $Heap = [Loc \rightarrow Val]$
- $Frames = Env \times Stmts^*$

Semantic Domains

- $Env = [Vars \rightarrow Loc]$
- $Closures = Env \times Callback$
- Val contains primitive values and closures
- $Heap = [Loc \rightarrow Val]$
- $Frames = Env \times Stmts^*$

Semantic Domains

- $Env = [Vars \to Loc]$
- $Closures = Env \times Callback$
- Val contains primitive values and closures
- $Heap = [Loc \to Val]$
- $Frames = Env \times Stmts^*$

Semantic Domains

- $Env = [Vars \to Loc]$
- $Closures = Env \times Callback$
- Val contains primitive values and closures
- $Heap = [Loc \to Val]$
- $Frames = Env \times Stmts^*$

Semantic Domains

- $Env = [Vars \to Loc]$
- $Closures = Env \times Callback$
- Val contains primitive values and closures
- $Heap = [Loc \to Val]$
- $Frames = Env \times Stmts^*$

Configurations

$\langle G, H, S \rangle$, where

- $G \in Env$,
- H is the global heap,
- $S \in Frame^*$, i.e., $S = \langle \ell_1, S_1 \rangle \ldots \langle \ell_n, S_n \rangle$ for $i : 1, \ldots, n$ and represents the call stack.

  In a pair $\langle \ell, w \rangle$, ℓ is the local environment and w is the corresponding program to be executed.

Configurations

$\langle G, H, S \rangle$, where

- $G \in Env$,
- H is the global heap,
- $S \in Frame^*$, i.e., $S = \langle \ell_1, S_1 \rangle \dots \langle \ell_n, S_n \rangle$ for $i : 1, \dots, n$ and represents the call stack.

  In a pair $\langle \ell, w \rangle$, ℓ is the local environment and w is the corresponding program to be executed.

Configurations

$\langle G, H, S \rangle$, where

- $G \in Env$,
- H is the global heap,
- $S \in Frame^*$, i.e., $S = \langle \ell_1, S_1 \rangle \ldots \langle \ell_n, S_n \rangle$ for $i : 1, \ldots, n$ and represents the call stack.

  In a pair $\langle \ell, w \rangle$, ℓ is the local environment and w is the corresponding program to be executed.

Let Declarations

$$\frac{\ell' = \ell[x/l],\ \ l_H(e) = v,\ \ H' = H[l/v],\ \ l \notin dom(H)}{\langle G, H, \langle \ell, let\ x = e\ in\ B\rangle \cdot S\rangle \rightarrow_L \langle G, H', \langle \ell', B\rangle \cdot S\rangle}$$

ℓ_H combines local environment ℓ and heap H

Let Declarations

$$\frac{\begin{array}{c}\ell' = \ell[f_1/l_1, \ldots, f_k/l_k],\ H' = H[l_1/\langle \ell, \lambda\vec{x}_1.P_1\rangle, \ldots, l_k/\langle \ell, \lambda\vec{x}_k.P_k\rangle] \\ l_i \notin dom(H),\ l_i \neq l_j,\ for\ i,j : 1,\ldots,k,\ i \neq j\end{array}}{\langle G, H, \langle \ell, let\ f_1 = \lambda\vec{x}_1.P_1, \ldots, f_k = \lambda\vec{x}_k.P_k\ in\ B\rangle \cdot S\rangle \rightarrow_L \langle G, H', \langle \ell', B\rangle \cdot S\rangle}$$

We adopt static binding as in Javascript
We use locations to access variables declared in outermost scopes
(an environment is an ordered lists of substitutions)

Observations

$$\frac{}{\langle G, H, \langle \ell, obs(e) \cdot B \rangle \cdot S \rangle \rightarrow_L^{\widehat{\ell_H}(e)} \langle G, H, \langle \ell, B \rangle \cdot S \rangle}$$

Store on Global Variables

$$\frac{x \notin dom(\ell) \quad G \cdot \ell_H(e) = w \neq \lambda\vec{y}.e}{\langle G, H, \langle \ell, store(x,e) \cdot B\rangle \cdot S\rangle \rightarrow_L \langle G[x/w], H, \langle \ell, B\rangle \cdot S\rangle}$$

Store on Local Variables

$$\frac{x \in dom(\ell) \quad \ell_H(e) = w \neq \lambda\vec{y}.e \quad \ell(x) = l}{\langle G, H, \langle \ell, store(x, e) \cdot B\rangle \cdot S\rangle \rightarrow_L \langle G, H[l/w], \langle \ell, B\rangle \cdot S\rangle}$$

Synchronous call

$$\frac{\begin{array}{c}\ell_H(f) = \langle \ell', \lambda\vec{y}.u\rangle,\ \ G \cdot (\ell_H) \cdot (\ell'_H)(\vec{v}) = \vec{v'},\ \ H' = H[\vec{l}/\vec{v'}],\ \ \ell'' = \ell[\vec{y}/\vec{l}],\\ \textit{for}\ \ \vec{l} = l_1, \ldots, l_k,\ \ l_i \notin \textit{dom}(H),\ \ l_i \neq l_j,\ \ \textit{for}\ \ i,j : 1, \ldots, k,\ \ i \neq j\end{array}}{\langle G, H, \langle \ell, f(\vec{v}) \cdot B\rangle \cdot S\rangle \rightarrow_L \langle G, H', \langle \ell'', u\rangle \cdot \langle \ell, B\rangle \cdot S\rangle}$$

Absorbtion

$$\frac{}{\langle G, H, \langle \ell, \epsilon \rangle \cdot S \rangle \rightarrow_L \langle G, H, S \rangle}$$

Plan

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation op and registers the callback cb to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of op, to the callback cb.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation op and registers the callback cb to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of op, to the callback cb.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation op and registers the callback cb to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of op, to the callback cb.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation *op* and registers the callback *cb* to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of *op*, to the callback *cb*.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation *op* and registers the callback *cb* to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of *op*, to the callback *cb*.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Instructions

- *reg*(e, u): registers callbacks in the word (list) $w \in F^*$ for event e, we use a list since we the callbacks must be processed in order.
- *call*(op, cb): invokes an asynchronous operation *op* and registers the callback *cb* to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of *op*, to the callback *cb*.
- *nexttick*$(f, \vec{v})$: enqueues the call to f with parameters $\vec{v}$ in the nextTick queue.
- *setimmediate*$(f, \vec{v})$: postpones the call to function f with parameters $\vec{v}$ to the next tick of the event loop.
- *trigger*$(e, \vec{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\vec{v}$.
- *unreg*(e, P): unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e,

Additional Notation

We now introduce an abstract machine to describes the semantics of the "event loop"'

- $Events = Events_i \cup Events_e$ is a finite set of (internal/external) event labels
- $Call_F$ is the set of callback calls $\{f(\vec{v}) | f \in F, \vec{v} \in Val^k, k \geq 0\}$.
- $Call_A$ is the set of asynchronous calls $\{call(a, cb) | a \in A, cb \in F\}$, where A is a set of labels.

Additional Notation

We now introduce an abstract machine to describes the semantics of the "event loop"'

- $Events = Events_i \cup Events_e$ is a finite set of (internal/external) event labels
- $Call_F$ is the set of callback calls $\{f(\vec{v}) | f \in F, \vec{v} \in Val^k, k \geq 0\}$.
- $Call_A$ is the set of asynchronous calls $\{call(a, cb) | a \in A, cb \in F\}$, where A is a set of labels.

Additional Notation

We now introduce an abstract machine to describes the semantics of the "event loop"'

- $Events = Events_i \cup Events_e$ is a finite set of (internal/external) event labels
- $Call_F$ is the set of callback calls $\{f(\vec{v}) | f \in F, \vec{v} \in Val^k, k \geq 0\}$.
- $Call_A$ is the set of asynchronous calls $\{call(a, cb) | a \in A, cb \in F\}$, where A is a set of labels.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, where

- $G \in Env$,
- $H \in Heap$,
- $E \in Listener$,
- $S \in Frame^*$,
- $C, Q, P \in (Env \times Call_F)^*$,
- $R \in (Env \times Call_A)^{\otimes}$.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle, \ldots$

- C is the (nexttick) queue of pending callback invocations generated by *nexttick*.
- Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events.
- P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*.
- R models the thread pool executing asynchronous operations

Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R\rangle, \ldots$

- C is the (nexttick) queue of pending callback invocations generated by *nexttick*.
- Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events.
- P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*.
- R models the thread pool executing asynchronous operations

  Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, ...

- C is the (nexttick) queue of pending callback invocations generated by *nexttick*.
- Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events.
- P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*.
- R models the thread pool executing asynchronous operations

  Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, ...

- C is the (nexttick) queue of pending callback invocations generated by *nexttick*.
- Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events.
- P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*.
- R models the thread pool executing asynchronous operations

  Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

Event Loop Configurations

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R \rangle$, ...

- C is the (nexttick) queue of pending callback invocations generated by *nexttick*.
- Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events.
- P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*.
- R models the thread pool executing asynchronous operations

  Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

Transitions in the Host Language

$$\frac{\langle G, H, S\rangle \rightarrow^{\alpha}_{L} \langle G', H', S'\rangle}{\langle G, H, E, S, C, Q, P, R\rangle \rightarrow^{\alpha} \langle G', H', E, S', C, Q, P, R\rangle}$$

Callback Registration

$$\frac{E' = E[evt/(E(evt) \cdot \langle \ell, u \rangle)]}{\langle G, H, E, \langle \ell, reg(evt, u) \cdot w \rangle \cdot S, C, Q, P, R \rangle \rightarrow \langle G, H, E', \langle \ell, w \rangle \cdot S, C, Q, P, R \rangle}$$

Registration Cancelation

$$\frac{E' = E[evt/(E(evt) \ominus u)]}{\langle G, H, E, \langle \ell, unreg(evt, u) \cdot w \rangle \cdot S, C, Q, P, R \rangle \rightarrow \langle G, H, E', \langle \ell, w \rangle \cdot S, C, Q, P, R \rangle}$$

Event Triggering

$$\frac{\begin{array}{c} evt \in Events_i \quad E(evt) = \langle \ell_1, u_1 \rangle \ldots \langle \ell_m, u_m \rangle \quad u_i = p^i_1 \cdot \ldots \cdot p^i_{k_i} \ \ for \ \ i : 1, \ldots, m \\ r = \langle \ell_1, p^1_1(\vec{v}) \rangle \cdot \ldots \cdot \langle \ell_1, p^1_{k_1}(\vec{v}) \rangle \ldots \langle \ell_m, p^m_1(\vec{v}) \rangle \cdot \ldots \cdot \langle \ell_m, p^m_{k_m}(\vec{v}) \rangle \quad \vec{v} \in Val^k \end{array}}{\langle G, H, E, \langle \ell, trigger(evt, \vec{v}) \cdot w \rangle \cdot S, C, Q, P, R \rangle \rightarrow \langle G, H, E, \langle \ell, w \rangle \cdot S, C, Q \cdot r, P, R \rangle}$$

Asynchronous Call

$$\frac{R' = R \oplus \{\langle \ell, call(a, cb) \rangle\}}{\langle G, H, E, \langle \ell, call(a, cb) \cdot w \rangle \cdot S, C, Q, P, R \rangle \rightarrow \langle G, H, E, \langle \ell, w \rangle \cdot S, C, Q, P, R' \rangle}$$

Termination of Async. Call

$$\frac{u = \langle \ell, cb(\vec{v}) \rangle \quad \vec{v} \in Val^k \quad R' = R \setminus \{\langle \ell, call(a, cb) \rangle\}}{\langle G, H, E, S, C, Q, P, R \rangle \rightarrow \langle G, H, E, S, C, Q \cdot u, P, R' \rangle}$$

External Event Triggering

$$\frac{\begin{array}{c} evt \in Events_e \quad E(evt) = \langle \ell_1, u_1 \rangle \ldots \langle \ell_m, u_m \rangle \quad u_i = p^i_1 \cdot \ldots \cdot p^i_{k_i} \text{ for } i : 1, \ldots, m \\ r = \langle \ell_1, p^1_1(\vec{v}) \rangle \cdot \ldots \cdot \langle \ell_1, p^1_{k_1}(\vec{v}) \rangle \ldots \langle \ell_m, p^m_1(\vec{v}) \rangle \cdot \ldots \cdot \langle \ell_m, p^m_{k_m}(\vec{v}) \rangle \quad \vec{v} \in Val^k \end{array}}{\langle G, H, E, S, C, Q, P, R \rangle \rightarrow \langle G, H, E, S, C, Q \cdot r, P, R \rangle}$$

Nexttick

$$\frac{G \cdot \ell_H(\vec{v}) = \vec{v'}}{\langle G, H, E, \langle \ell, nextT(f, \vec{v}) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E, \langle \ell, w\rangle \cdot S, C \cdot \langle \ell, f(\vec{v'})\rangle, Q, P, R\rangle}$$

Setimmediate

$$\frac{G \cdot \ell_H(\vec{v}) = \vec{v'}}{\langle G, H, E, \langle \ell, setI(f, \vec{v}) \cdot w \rangle \cdot S, C, Q, P, R \rangle \rightarrow \langle G, H, E, \langle \ell, w \rangle \cdot S, C, Q, P \cdot \langle \ell, f(\vec{v'}) \rangle, R \rangle}$$

Selection from Nexttick Queue

$$\frac{\begin{array}{c}\ell_H(p) = \langle \ell', \lambda\vec{y}.s\rangle,\ \ G \cdot (\ell_H) \cdot (\ell'_H)(\vec{v}) = \vec{v'},\ \ H' = H[\vec{l}/\vec{v'}],\ \ \ell'' = \ell[\vec{y}/\vec{l}], \\ \mathit{for}\ \ \vec{l} = l_1, \ldots, l_k,\ \ l_i \notin \mathit{dom}(H),\ \ l_i \neq l_j,\ \ \mathit{for}\ \ i,j : 1, \ldots, k,\ \ i \neq j\end{array}}{\langle G, H, E, \perp, \langle \ell, p(\vec{v})\rangle \cdot C, Q, P, R\rangle \rightarrow \langle G, H', E, \langle \ell', s\rangle, C, Q, P, R\rangle}$$

Selection from Poll Queue

$$\frac{\begin{array}{c}\ell_H(f) = \langle \ell', \lambda\vec{y}.s\rangle,\ \ G \cdot (\ell_H) \cdot (\ell'_H)(\vec{v}) = \vec{v'},\ \ H' = H[\vec{l}/\vec{v'}],\ \ \ell'' = \ell[\vec{y}/\vec{l}], \\ \mathit{for}\ \ \vec{l} = l_1, \ldots, l_k,\ \ l_i \notin \mathit{dom}(H),\ \ l_i \neq l_j,\ \ \mathit{for}\ \ i, j : 1, \ldots, k,\ \ i \neq j\end{array}}{\langle G, H, E, \bot, \epsilon, p(\vec{v}) \cdot Q, P, R\rangle \rightarrow \langle G, H', E, \langle \ell', s\rangle, \epsilon, Q, P, R\rangle}$$

Selection from Pending Queue

$$\frac{}{\langle G, H, E, \perp, \epsilon, \epsilon, P, R\rangle \rightarrow \langle G, H, E, \perp, \epsilon, P, \epsilon, R\rangle}$$

Plan

Simple Node Example

```
var fs = require('fs');
fs.readFile('input.txt', function cb (data) {
console.log(data.toString());
});

console.log('Program Ended');
```

$$
\begin{tikzcd}[column sep=small]
& S = call(readFile, cb) \cdot log(\ldots) \quad Q = \epsilon \quad R = \emptyset \arrow[d] & \\
& S = log(\ldots) \quad Q = \epsilon \quad R = \{\!\{cb\}\!\} \arrow[dl] \arrow[dr] & \\
S = log(\ldots) \quad Q = \langle cb, \vec{v} \rangle \quad R = \emptyset \arrow[dr] & & S = \perp \quad Q = \epsilon \quad R = \{\!\{cb\}\!\} \arrow[dl] \\
& S = \perp \quad Q = \langle cb, \vec{v} \rangle \quad R = \emptyset \arrow[d] & \\
& S = log(\ldots) \quad Q = \epsilon \quad R = \emptyset \arrow[d] & \\
& S = \perp \quad Q = \epsilon \quad R = \emptyset &
\end{tikzcd}
$$

$cb = \lambda data.log(data)$

Conclusions

- Event-driven programs have a non-deterministic behavior: difficult to program and to verify
- The abstract machine can be used to understand the behavior, apply analysis and verification techniques
- Starting from this model: Js promises, bounded model checking, decidable fragments (?)
- Tools like Loupe[2] can be written

[2] latentflip.com/loupe/

References

M. Emmi, P. Ganty, R. Majumdar and F. Rosa-Velardo
Analysis of Asynchronous Programs with Event-Based Synchronization
Springer-Verlag Berlin Heidelberg 2015
J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 535–559, 2015

G. Geeraerts, A. Heußner, J.-F. Raskin
On the Verification of Concurrent, Asynchronous Programs with Waiting Queues
ACM Trans. Embedd. Comput. Syst. 0, 0, Article 0 (2013), 25 pages.

S. Alimadadi et al.
Understanding JavaScript Event-Based Interactions with Clematis
ACM Trans. Softw. Eng. Methodol. 2015

P. Roberts, What the heck is the event loop anyway?
JSConf EU 2014 (youtube video)

https://nodejs.org/en/docs/