The Resumption Monad Transformer and its Implementation in JavaScript

Diploma Thesis Presentation

Georgios Sakkas

Supervisor: Nikolaos S. Papaspyrou, Associate Professor NTUA

July 6, 2018

National Technical University of Athens
School of Electrical and Computer Engineering

Introduction

Introduction

Resumptions are a valuable tool in the analysis and design of semantic models for concurrent programming languages, in which computations consist of **sequences of atomic steps** that may be interleaved.

In this work we define a Resumption Monad Transformer (RMT) in **JavaScript** and we investigate how this can be a *low-overhead* and extremely *modular* way to define the **denotational semantics** of a simple imperative language, which has side-effects and supports concurrency.

Monads and Monad Transformers

Monads

A monad is essentially a triple $\langle M, \mathbf{unit}_M, \mathbf{bind}_M \rangle$ consisting of a type constructor M and a pair of polymorphic functions that must satisfy the *three monad laws*. In *functional* we have:

$$\texttt{return} :: a \to M\ a \qquad (\mathbf{unit}_M)$$

$$\texttt{>>=} :: M\ a \to (a \to M\ b) \to M\ b \qquad (\mathbf{bind}_M)$$

Types constructed by monad M denote **computations**.

- The type $M\ a$ denotes computations returning values of type a.
- The result of `return` v is a computation returning the value v.
- The result of m `>>=` f is the combined computation of m, returning v, followed by computation $f\ v$.

MultiMonads and StrongMonads

It is also useful to distinguish **two subclasses** of monads with additional features.

```
class Monad m => MultiMonad m where
(+|+) :: m a -> m a -> m a

class Monad m => StrongMonad m where
(+:+) :: m a -> m b -> m (a, b)
```

- `+|+` indicates an **option** between two alternative computations.
- `+:+` indicates a **combination** of two simultaneous computations.
- Their exact behavior **depends** on a monad's definition.

Monad Transformers

- Monad transformers are similar to regular monads, but they are not standalone entities: instead, they modify the behavior of an underlying monad.

- Monad transformers are mappings between monads and they are implemented as *higher-order* type constructors of kind $(\star \to \star) \to \star \to \star$.

- The intuition behind them is that, if T is a monad transformer and m is a monad, then $T\ m$ is also a monad and its properties are defined in terms of the properties of m.

States and State Monads

Program State

- The notion of state is a very important one in the study of the impure languages.
- A state is an element of a type which supports two main operations, `load` and `store`, for retrieving and updating the contents of a variable in memory.
- A distinguished element of this type is the `initial` state, typically a state with all variables uninitialized.

State Monad

- A class of monads that are aware of the state is also useful. Therefore we need a state monad.
- Class `StateMonad` supports two operations as an interface between computations and the state.

```
class Monad m => StateMonad s m where
  setState :: (s -> s) -> m s
  getState :: m s
  getState = setState id
```

State Monad Transformer

- A monad transformer `D s` can be defined as follows.
- Parameter `m` specifies the monad representing the stateless computations.

```
newtype D s m a = D (s -> m (a, s))

instance Monad m => Monad (D s m) where
  return v  = D (\s -> return (v, s))
  D r >>= f = D (\s -> r s >>= \(v', s') ->
                let D r' = f v' in r' s')
```

State Monad Transformer

- Monads constructed using `D` are **aware of the state**.
- Monad `D s m` is an instance of `StateMonad` for state type `s`.

```
instance Monad m => StateMonad s (D s m) where
  setState f = D (\s -> return (s, f s))
```

- With the *identity monad* `Id` for stateless computations, we end up with the conventional direct semantics monad M.

```
type M a = D S Id a
```

Resumptions

Resumptions

- Resumptions are constructs which split a computation in a single atomic step (to be executed first) and a **resumed** part, which corresponds to the rest of the computation.
- Resumptions can model interleaved computations and therefore are a denotational model of *concurrency*.
- So, resumptions can be used in a monadic style to define the semantics of concurrent programming languages.

Traces and Interleaving

- A natural model of concurrency is the trace model.
- Threads are (potentially infinite) streams of atomic operations.
- The meaning of concurrent thread execution defined as the set of all their possible thread interleavings.

Interleaving Example

- For example, two simple **threads** $a = [a_0, a_1]$ and $b = [b_0]$, where a_0, a_1 , and b_0 are **atomic operations**.
- The **concurrent** execution of threads a and b, $a \parallel b$, is denoted by the set of all their possible interleavings.
- **traces**$(a \parallel b) = \{[a_0, a_1, b_0], [a_0, b_0, a_1], [b_0, a_0, a_1]\}$

Resumptions

The most basic resumption monad contains only a notion of sequencing atomic steps and nothing else:

```
data R a = Computed a | Resume (R a)
```

The resumption monad must have:

```
instance Monad R where
  return               = Computed
  (Computed v) >>= f = f v
  (Resume r) >>= f   = Resume (r >>= f)
```

Resumptions

- **Resumptions** not enough for imperative languages.
- **States** must be introduced to allow side-effects.
- Define a **resumption monad transformer** that can be used to ”lift” a **state monad**.

Resumption Monad Transformers

The resumption monad transformer is defined similarly as:

```
data R m a = Computed a | Resume (m (R m a))
```

For the resumption monad transformer we have:

```
instance Monad m => Monad (R m) where
  return             = Computed
  (Computed v) >>= f = f v
  (Resume m) >>= f   = Resume (m >>= \r ->
                              return (r >>= f))
```

Resumption Monad Transformers

- Parameter `m` of RMT is a monad representing **computations**.
- A computation of type `R m a` is either a **computed** value of type a or a computation of type `m (R m a)`, which **produces a resumption**, just like resumptions.
- A version of monad `M` which allows **interleaved computations** can be defined by applying `R` to the direct semantics monad.

```
type M a = R (D S Id) a
```

- Two functions to **convert between computations** of type `R m a` and `m a` are needed.

RMT Additional Operations

The first fully evaluates a resumption by performing all atomic steps.

```
run :: Monad m => R m a -> m a
run (Computed v) = return v
run (Resume m)   = m >>= run
```

The second produces a computation with just one atomic step.

```
step :: Monad m => m a -> R m a
step m = Resume (m >>= (return ∘ Computed))
```

RMTs and Interleaving

Returning to the trace model with the two threads $a = [a_0, a_1]$ and $b = [b_0]$, we have:

1. $\textit{Resume}\ (a_0$ **>>=** **return** $(\textit{Resume}\ (a_1$ **>>=** **return** $($
   $\textit{Resume}\ (b_0$ **>>=** **return** $(\textit{Computed}\ ())))))$
2. $\textit{Resume}\ (a_0$ **>>=** **return** $(\textit{Resume}\ (b_0$ **>>=** **return** $($
   $\textit{Resume}\ (a_1$ **>>=** **return** $(\textit{Computed}\ ())))))$
3. $\textit{Resume}\ (b_0$ **>>=** **return** $(\textit{Resume}\ (a_0$ **>>=** **return** $($
   $\textit{Resume}\ (a_1$ **>>=** **return** $(\textit{Computed}\ ())))))$

Here, **>>=** and **return** are the *bind* and *unit* operations of the state monad.

RMTs in JavaScript

RMTs in JavaScript

- Every monad is defined as a `class` in JavaScript.
- Monad transformers are *functions* that take a *monad **m*** as an argument and return a new `class` that defines the **new** monad.
- The "monad" classes have a *constructor*, a **static** method *unit* and a method *bind*.

RMTs in JavaScript

For example the implementation of the simple Identity monad:

```
class IdentityM {
    constructor(x) { this.valueId = x; }
    static unit(x) { return new IdentityM(x); }
    bind(f)        { return f(this.valueId); }
}
```

RMTs in JavaScript

The implementation of the Resumption Monad Transformer:

```
function ResumptionT(M) {
    return class RM {
        constructor(computed, Mnd, a) {
            // true -> "Computed", false -> "Resume"
            this.status = computed;
            this.Mnd = Mnd;
            this.value = a;
        }
        ...
    }
}
```

RMTs in JavaScript

```
function ResumptionT(M) {
    return class RM {
        ...
        static unit(x) { return Computed(x); }
        bind(f) {
            if (this.status)
                return f(this.value);
            else
                return Resume(this.Mnd.bind(r =>
                            M.unit(r.bind(f))));
        }
        ...
    }
}
```

RMTs in JavaScript

```
function ResumptionT(M) {
    return class RM {
        ...
        runR() {
            if (this.status)
                return M.unit(this.value);
            else
                return this.Mnd.bind(r => r.runR());
        }

        static stepR(Mnd) {
            return Resume(Mnd.bind(x => M.unit(RM.unit(x))));
        }
    }
}
```

A modular semantics of concurrency

Semantics of Concurrency

Consider the simple sequential imperative language:

$$s \ ::= \ x \textbf{ := } e \mid s \textbf{ ; } s \mid \textbf{if } e \textbf{ then } s \textbf{ else } s \mid \textbf{while } e \textbf{ do } s$$

The language of expressions e is the following.

$$e \ ::= \ x \mid e + e \mid e * e \mid \ldots \mid x{+}{+} \mid \ldots \mid e < e \mid e == e \mid \ldots$$

The **semantic function** is straightforward. For example:

$[\![\,]\!]$ `::` $[\![s]\!]$ `-> M a`

$[\![x$ **`:=`** $e]\!]$ `=`
  $[\![e]\!]$ **`>>=`** `\n -> setState (store i n)` **`>>=`** `\s ->`
    `return` `n`

$[\![$**`if`** e **`then`** s_1 **`else`** $s_2]\!]$ `=`
  $[\![e]\!]$ **`>>=`** `\c ->` **`if`** `c` **`then`** $[\![s_1]\!]$ **`else`** $[\![s_2]\!]$

$[\![e_1 + e_2]\!]$ `=`
  $[\![e_1]\!]$ **`>>=`** `\`v_1 `->` $[\![e_2]\!]$ **`>>=`** `\`v_2 `->` **`return`** v_1+v_2

Semantics of Concurrency

Let us now introduce **concurrency** in our language:

$$s \; ::= \; \ldots \mid s \parallel s \mid \langle s \rangle$$

The semantic function is:

$[\![s_1 \parallel s_2]\!]$ `=`
  $[\![s_1]\!]$ `+:+` $[\![s_2]\!]$ `>>= \p -> return ()`

$[\![\langle s \rangle]\!]$ `=`
  `step (run` $[\![s]\!]$`)`

Benchmarks

Our benchmark algorithms include:

1. **sieve**: The simple algorithm for the sieve of Eratosthenes
2. **pi**: A pi approximation algorithm
3. **primality**: A simple primality test algorithm
4. **insert**: Insertion sort
5. **reduce**: The reduction of a given array
6. **mat-vec**: Matrix-vector multiplication
7. **comb**: Enumerating all possible combinations (m comb n)

Benchmarks

1. Each benchmark was executed 100 in the NodeJS framework for the *sequential* and the *concurrent* tests separately.
2. The average *execution times* were measured for each benchmark.
3. As a metric of our performance, we use the overhead that the concurrent execution has.
4. A *baseline* case is needed to compare results.

JavaScript Promises

Promises

- JavaScript Promises have similar semantics to resumptions.
- Promises are used for asynchronous computations, but are used as a concurrent model in JavaScript.
- Three mutually exclusive states: pending, fulfilled and rejected.

Promises

If e is a Promise object then the following operations for working with Promises are defined:

- *Promise*() creates a **new Promise** object e.
- *Promise.resolve*(e_2) resolves a Promise e_1 to the **value** of e_2.
- e_1.*then*(e_2) schedules Promise e_2 to be **executed after** the Promise e_1 is resolved.
- *Promise.all*([e_i]), where [e_i] is an iterable of Promises e_i, creates a new Promise object e which is **resolved when all** of the iterable's Promises are resolved.

Results

Benchmark time averages

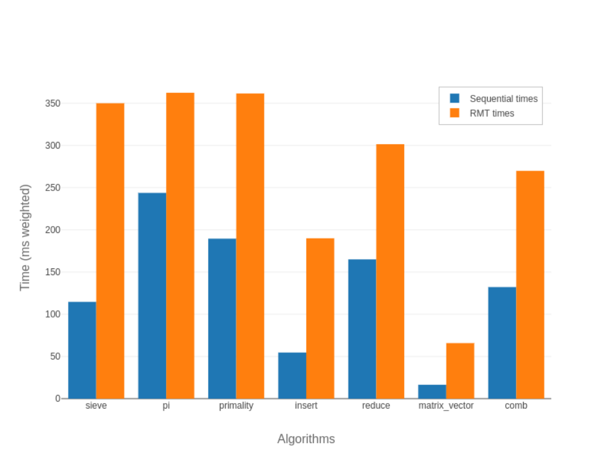

Figure 1: Weighted run-time averages by algorithm time complexity

Benchmark overhead averages

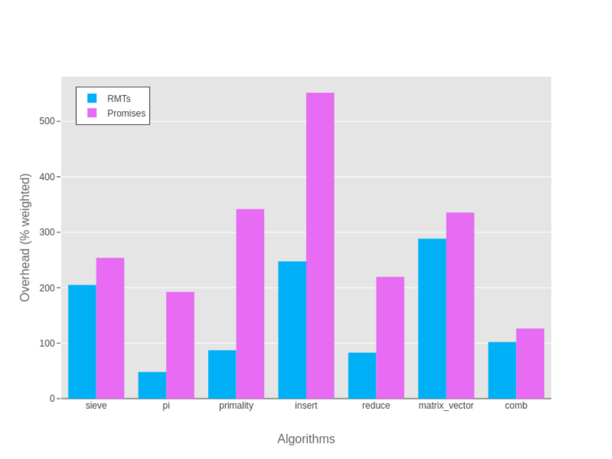

Figure 2: Weighted overheads averages by algorithm time complexity

Largest inputs

Benchmark	Method	Rank of input									
		1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
sieve	RMT	**198.78**	**204.33**	**212.94**	**212.19**	**217.04**	**201.24**	**205.64**	**202.45**	**207.22**	**210.19**
	Prom.	203.82	241.00	243.75	268.17	255.58	292.53	294.35	301.67	327.87	336.59
pi	RMT	**50.87**	**50.75**	**51.69**	**49.87**	**51.66**	**50.00**	**50.28**	**51.00**	**50.94**	**51.18**
	Prom.	197.89	156.15	136.10	104.25	106.44	119.67	124.86	155.10	153.43	191.92
primality	RMT	**54.45**	**82.96**	**71.36**	**76.47**	**83.60**	**83.26**	**81.84**	**90.48**	**94.85**	**95.54**
	Prom.	313.10	327.19	322.14	350.54	341.14	345.28	351.66	332.46	351.79	341.01
insert	RMT	**249.43**	**247.53**	**246.27**	**238.66**	**235.99**	**241.69**	**233.94**	**245.79**	**245.84**	**249.87**
	Prom.	560.55	573.52	579.80	590.20	576.71	575.68	593.31	587.44	476.16	411.09
reduce	RMT	**80.03**	**86.20**	**81.78**	**82.41**	**81.01**	**83.87**	**82.19**	**81.74**	**83.03**	**82.81**
	Prom.	140.00	132.81	182.28	187.10	189.63	188.00	216.62	225.71	240.63	228.59
mat-vec	RMT	**272.79**	304.81	**294.50**	**276.67**	**301.52**	321.10	**330.15**	316.46	**342.87**	**331.45**
	Prom.	313.98	**265.17**	315.90	358.57	357.06	**298.53**	398.56	**299.30**	402.65	381.15
comb	RMT	**30.01**	**82.59**	**67.23**	**66.20**	**82.82**	**76.30**	**72.21**	**117.01**	**104.80**	**103.80**
	Prom.	63.65	129.30	158.25	87.44	90.58	94.78	105.20	117.22	134.19	131.13

Table 1: Overheads for the ten biggest inputs for every benchmark tested

Average overheads per input

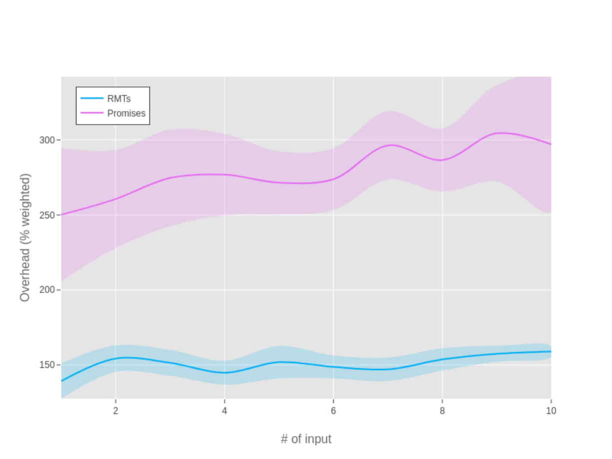

Figure 3: Average overheads for the ten biggest inputs of each benchmark

Conclusions

Conclusions

- The *resumption monad transformer* implementation outperformed Promises.
- *Resumption monad transformers* can a low-overhead and extremely modular constructs for the semantics of concurrency.
- Generator functions can be used to import laziness in `bind` functions in the future .

Thank you!

Any questions?