

ktulu references

Jean-Louis Villecroze

jlrv@fizz.org @CocoaGeek

Preview 11
October 18, 2020

Module

The `modCTM` module implements a couple of *elemental* objects that supports the development of *ktulu* as an extension of *fizz*. To be available the module must be loaded in the *runtime*, either with the `load` command or by specifying the module's name when using a JSON solution file. For example, if we wanted to execute a test defined in a *fizz* file `foo.fizz`, we would create a `foo.json` file (in the same folder) with the following content:

```
1 {  
2   "solution" : {  
3     "modules" :  ["modCTM"],  
4     "sources" :  [  
5       "foo.fizz"  
6     ],  
7     "globals" :  []  
8   }  
9 }
```

We can then start *fizz* with the JSON file as a command line argument, and it will load the module as well as the *fizz* file (and any other specified file):

```
jlrv@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/foo.json  
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]  
Press the ESC key at anytime for input prompt  
  
load : loading ./etc/ktulu/tests/foo.json ...  
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s  
load : loading ./etc/ktulu/tests/foo.fizz ...  
load : loaded ./etc/ktulu/tests/foo.fizz in 0.014s  
load : loading completed in 0.015s
```

Elemental classes

The implementation of the *Conscious Turing Machine* as proposed by Manuel, Lenore and Avrim Blum in their article, is built around three core *elementals*: `CTMCBFSActor`, `CTMCFUNActor` and `CTMCStage`.

Unlike in the article, the implementation differentiate between *chunks* by having four possible type of *chunks* for practical reasons:

tweet	a declaration from a <i>processor</i> .
query	a question from a <i>processor</i> in need of replies.
reply	a reply from a <i>processor</i> to another <i>processor's query</i> .
kudos	a feedback to a <i>processor's reply</i> to a <i>query</i> .

A *chunk* in this implementation also differs slightly from the article. It is represented as a *list term* with an arity of 8 and contains:

[originator,tick,unique id,type,gist,weight,intensity,mood]

CTMCBFSActor

This *elemental* implements the part of a *processor* that communicate with the *stage*: receives STM broadcast and supplies *chunks* for competition. The class is derived from the `MRKCBFSolver` and as such can handle logical queries.

It supports the following *properties*:

stage	a <i>symbol</i> that indicate the <i>CTM</i> to which the actor is connected to. the default is <code>ctm</code> .
dreco	the DROP operator recovery time (in ticks), default is 0 (the actor won't respect the operator).
ctime	how long (in ticks) to keep track of chunks sent and received. If no custom <code>ttl</code> value is given to the <i>elemental</i> , a custom value will be set based on <code>ctime</code> and the stage's tick.
wtmod	initial value of the weight modifier based on feedback.
tlink	kudos threshold for linking (default is 5).
links	set to <code>yes</code> to allow the actor to establish direct link with other actors. If links are already established, set to a <i>list</i> that contains <i>lists</i> , each providing: the <i>functor</i> to be used to match patterns, the label of the processors and the last linking value.
rtime	how long (in ticks) to keep track of the content of the STM for. Default is 0.

The following callback *queries* will be sent to the *elemental* in specific situations:

(init)	queried when the <i>elemental</i> is attached to the <i>runtime</i> environment. This only happens once.
(beat, :t)	queried regularly based on the runtime setting. The second <i>term</i> is the last received tick.
(tick, :t, :l, :a)	queried for every broadcast from the <i>stage</i> . The second <i>term</i> is the tick number, the third is the content of the STM (as a <i>list</i>) and the fourth. is a <i>frame</i> holding ancillary data.
(link, :t, :l)	queried for every query sent directly by another processor. The second <i>term</i> is the tick number at the time of reception and the third is the chunk (in a <i>list</i>).

The following *logical queries* are available to support posting chunks (they can be called from within the *elemental* it-self or from another):

(tweet, :f, [:g, :w, :m])	submit a <i>tweet</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
(query, :f, [:g, :w, :m], :r)	submit a <i>query</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
(reply, :u, :f, [:g, :w, :m])	submit a <i>reply</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
(kudos, :u, :f, [:g, :w, :m])	submit a <i>kudos</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .

with `:f` being a standing for a *frame* containing some options for the posting:

retries	the number of ticks for which the <i>actor</i> will automatically post the exact same <i>chunk</i> , until it is included in the STM. If the value is 0 (the default), there won't be any retry. If the value is -1 the actor will retry until the <i>chunk</i> is in the STM.
replace	if set to the value yes , this chunk will replace any previously and still tried posting. (only valid for <i>tweet</i> and <i>query</i>).
collect	if set to the value yes , the <i>actor</i> will prepend all <i>replies</i> to a <i>list</i> .
diverse	if set to the value yes , the <i>actor</i> will inforce that all <i>replies</i> are different before passing them on.
wide	if set to the value yes , the <i>actor</i> will also submit a query to the competition when it uses a direct link.
repeats	the number of tick for which the <i>actor</i> will automatically post the exact same <i>tweet</i> regardless of getting in the STM. If the value is 0 (the default), there won't be any repost. If the value is -1 the actor will keep posting the <i>tweet</i> until it is replaced.
boost	a factor to be applied to the weight of a chunk when repeating or retrying it. The factor won't be applied when repeats is set to -1.
timeout	when used for a query, this express the number of ticks to expect a reply for. When the timeout occurs, the value null will be returned by the <i>logical query</i> or will be prepended to the <i>list</i> when using the collect option.
dropout	when used for a query, this the amount of weight to take away from any direct links. Ideally, this value should be the same that the one that will be provided with a kudos. Once the weight of a direct link drops below a threshold, the link will no longer be used.

with **:u** being the *unique id* of either the *query chunk* when used to reply, or the *unique id* of the *reply chunk* when used to provide feedback with a *kudos chunks*.

with **:g** being a standing for any *term* that is the *gist* of the *chunk*

with **:w** being the weight of the *chunk* and **:m** being the mood of it.

In a case of a *query*, the *predicate* takes in a fourth *term* which will be unified with any *chunk* received as *reply* (or with a *list* of *chunks* when using **collect**).

To *recall* the content of the STM, as seen by an actor, at some moment in the past, the following *query* can be called upon:

(**recall**, {}, **:filter**, **:result**) Its third *term* is an optional filter on the content of the chunks. Thus to be of use it should be a *list* of arity 8. The fourth *term* will be unified with the *list* of matching *chunks*. The second *term* can be used to specify the option **own** (with the value **yes**) which will restrict the recall to only the chunks of the actor (including the one that failed to appears in the STM).

To *wait* for a specific chunk to be in the STM (or received directly), the following *query* can be called upon:

(**await**, {}, **:filter**, **:result**) Its third *term* is an optional filter on the content of the chunks. Thus to be of use it should be a *list* of arity 8. The fourth *term* will be unified with the *list* of matching *chunks*. The second *term* can be used to specify options.

The supported options are:

collect if set to the value **yes**, the *actor* will prepend all *chunk* to a *list*.
limit the maximum number of chunks to be provided by this query.
timeout the number of ticks to await for (0 for no timeout).

CTMCFUNActor

This *elemental*, implements a *processor* on top of a **FZZCFUNRunner** *elemental*. Most of what is described for **CTMCBFSActor** is valid for this *elemental* as well, except for the callbacks *queries*. Since this *elemental* is *imperative* based (rather than *logical*), instead of providing *prototypes* to handle the queries, you will need to define *functions*. For example:

```

1 ctm.actor.foo {
2   class      = CTMCFUNActor,
3   primitives = funx.primitive,
4 } {
5
6   (init,[],[ // function to handle the attachment of the elemental to the substrate
7     // your code here
8   ]);
9
10  (beat,[t],[ // function to handle the substrate's pulse
11    // your code here
12  ]);
13
14  (tick,[t,1,a],[ // function to handle CTM's ticks
15    // your code here
16  ]);
17
18 }
```

CTMCStage

This *elemental* implements the *stage*. Its purpose is to collect all *processor* supplied *chunks*, perform the *up-tree competition* then broadcast the content of its *STM* to all *processors*.

It supports the following *properties*:

mode the operation mode: **probabilistic**, **deterministic** or **weightcentric**
size the size of the STM (that is the number of *chunks* that will get broadcasted at each tick).
ttag accepted tick lag (in ticks), default is 0. If a *chunk* is delivered late it will be ignored.
tick tick frequency (ms) or the label of a ticker *elemental*
mood mood factor value.
drop DROP operator value.
intr interruption's intensity.
func competition function.
esel weight equality selector.

The *competition functions* currently supported are:

abs_weight use the absolute value of a chunk's weight.
intensity use the value of a chunk's intensity.
intensity_mood use the sum of the chunk's intensity with its mood.
abs_mood use the absolute value of a chunk's mood.
intensity_mood_2 use the sum of the chunk's intensity with its mood halved.

The *weight equality selectors* currently supported are:

left	pick the <i>chunk</i> on the left (of the tree node).
right	pick the <i>chunk</i> on the right (of the tree node).
toss	pick one of the <i>chunk</i> randomly.

At every ticks, the *stage* broadcasts out the content of the STM by publishing a *statement* into the *runtime*. The label of the *statement* is the label of the *elemental* with the postfix `.tick`. The terms in the statement are: the tick number, a *list* with the STM content and a *frame* containing some details on the CTM:

drop	the DROP operator value (only if when deterministic).
intr	interruption's intensity.
chunks	the total number of chunks received by the CTM for this tick.
time	the time of the tick.
mood	mood factor value.
tick	the elapsed tick value.

A simple example

Let's briefly look at how to setup a very simple test, where two *processors* are competing in a *deterministic CTM*.

First, create a new *fizz* file called `hello.fizz`. In it, we will first provide the definition of the *stage*:

```

1 ctm {
2
3     class    = CTMCStage,
4
5     mode     = deterministic,
6     size     = 1,
7     tick     = 250,
8     drop     = 0.5,
9     func     = abs_weight,
10    esel     = toss
11
12 }
```

The content of the *STM* will get broadcasted 4 times per seconds and the up-tree competition will use the absolute value of each *chunks'* weight. If two *chunks* have the same result for the *chunk function*, the *stage* will randomly pick.

Create now a file `hello.json` to setup the solution:

```

1 {
2     "solution" : {
3         "modules" :    ["modCTM"],
4         "sources" :    [
5             "hello.fizz"
6         ],
7         "globals" :    []
8     }
9 }
```

Before trying this out, let's add another *elemental* definition in `hello.fizz` so that we can see the *STM* content at each broadcast:

```

1 ctm.obs {
2
3     () :- @ctm.tick(:t,:l,_),
4           console.puts(:t," ",:l);
5
6 }

```

Note here, that the *elemental* `ctm.obs` is not considered a *processor*, but takes advantage of the fact that the content of the *STM* is broadcasted as a logical *statement* within the *runtime*, it thus can be used as a *trigger predicate*.

We can now, launch *fizz* with the solution file and observe the content of the STM:

```

jlv@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.002s
load : loading completed in 0.003s
1 []
2 []
3 []
4 []
5 []
6 []
7 []
8 []
9 []
10 []
11 []

```

Since there is no *processors* yet, the content of the *STM* is empty, as expected. Let's now add to `hello.fizz` a first *processor* which will *tweet* frequently, but not all the time. For this, we will use an *elemental* that generate *statements* every 0.5 seconds:

```

1 ctm.proc.tick {
2     class      = FZZCTicker,
3     tick       = 0.5
4 } {}

```

In the *processor*, we will use `ctm.proc.tick` as a *trigger predicate* to post a *tweet* some of the time:

```

1 ctm.proc.sheldon {
2     class      = CTMCBFSActor,
3     dreco      = 4,
4     weight     = 1,
5     mood       = 1
6 } {
7
8     () :- @ctm.proc.tick(_,_),

```

```

9         rnd.real(1,_[gt(0.3)]),
10         ~self(tweet,{},[bazinga,$weight,$mood]);
11
12 }

```

On line 9, we use the *primitive* `rnd.real` to draw a random number (between 0 and 1) and use it to only *tweet* 70% of the time. On the following line, we query the *elemental* itself to post a *tweet* which *gist* is the *symbol* `bazinga`. The weight and the mood of the *chunk* are taken from the *elemental* properties.

Let's try this:

```

jlv@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.003s
load : loading completed in 0.005s
1 []
2 []
3 []
4 []
5 [[ctm.proc.sheldon, 4, enkoogy, tweet, bazinga, 1, 1, 1]]
6 []
7 [[ctm.proc.sheldon, 6, vfobcwXu, tweet, bazinga, 0.500000, 0.500000, 1]]
8 []
9 [[ctm.proc.sheldon, 9, ksqygwio, tweet, bazinga, 0.875000, 0.875000, 1]]
10 []
11 [[ctm.proc.sheldon, 11, kyygvic, tweet, bazinga, 1, 1, 1]]
12 []
13 [[ctm.proc.sheldon, 13, jvhyrins, tweet, bazinga, 0.625000, 0.625000, 1]]
14 []
15 [[ctm.proc.sheldon, 15, hqvkskji, tweet, bazinga, 0.875000, 0.875000, 1]]
16 []

```

Note how the weight of the *chunk* change as the *processor* take into account the *DROP operator*. Let's now add a second *processor* in `hello.fizz`:

```

1 ctm.proc.leonard {
2     class  = CTMCBFSActor,
3     dreco  = 4,
4     weight = 1,
5     mood   = 1
6 } {
7
8     () :- @ctm.proc.tick(,_),
9           rnd.real(1,_[gt(0.2)]),
10           ~self(tweet,{},[oh_boy,$weight,$mood]);
11
12 }

```

It is setup pretty much the same way, except for the *gist* and for the likelihood of it posting a *tweet*. Let see

see how that plays out:

```
jlv@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.005s
load : loading completed in 0.006s
1 []
2 []
3 [[ctm.proc.leonard, 3, bhddympv, tweet, oh_boy, 1, 1, 1]]
4 []
5 [[ctm.proc.sheldon, 5, xxxguhyn, tweet, bazinga, 1, 1, 1]]
6 []
7 [[ctm.proc.leonard, 7, rcatiahh, tweet, oh_boy, 0.875000, 0.875000, 1]]
8 []
9 [[ctm.proc.leonard, 9, ctgcsqws, tweet, oh_boy, 1, 0.937500, 1]]
10 []
11 [[ctm.proc.leonard, 11, ehvyycet, tweet, oh_boy, 0.625000, 0.625000, 1]]
12 []
13 [[ctm.proc.sheldon, 12, featlusi, tweet, bazinga, 1, 0.875000, 1]]
14 []
15 [[ctm.proc.leonard, 14, bsmqfrkc, tweet, oh_boy, 1, 1, 1]]
16 []
17 [[ctm.proc.sheldon, 17, mvfhfpib, tweet, bazinga, 0.875000, 0.750000, 1]]
18 []
19 [[ctm.proc.sheldon, 19, nmsusiee, tweet, bazinga, 1, 0.937500, 1]]
20 []
21 [[ctm.proc.leonard, 21, avaqkgat, tweet, oh_boy, 1, 0.812500, 1]]
22 []
23 [[ctm.proc.leonard, 23, dsnkrrnp, tweet, oh_boy, 0.625000, 0.625000, 1]]
24 []
25 [[ctm.proc.leonard, 25, cllllvqk, tweet, oh_boy, 0.875000, 0.875000, 1]]
26 []
27 [[ctm.proc.leonard, 27, qkvqonjj, tweet, oh_boy, 1, 1, 1]]
28 []
29 [[ctm.proc.sheldon, 29, ejmdtabr, tweet, bazinga, 1, 0.812500, 1]]
30 []
```