An introduction to *Conscious Turing Machines* with *fizz*

Jean-Louis Villecroze
`jlv@f1zz.org @CocoaGeek`

June 16, 2019 (Revision 2)

Abstract

In this article, we will detail the implementation of a simplified *Conscious Turing Machine* as proposed by Manuel, Lenore and Avrim Blum. We will also build simple examples of such *machines* that will *consciously* memorize numbers or list of numbers and compute the sum of a list of numbers.

Acknowledgement

The author would like to thanks Manuel and Lenore Blum, for their renewed support and patience during the development of this work.

Prerequisite

A basic understanding of the concepts behind *fizz* (version 0.5 and up) is expected from the reader of this article. It is suggested to read the introductory article *Building a simple stock prices monitor with fizz* [1] first or at least read sections two to four of the *user manual* for an overview of the language and runtime. The complete source code discussed in this article can be downloaded from the author's web site[2].

What is a *Conscious Turing Machine*?

A *Conscious Turing Machine* (CTM) is a cognitive architecture inspired by cognitive science, where *consciouness* (experiencing *qualia*) is at the center of the architecture. While Professor Blum[3] drew from previous works by many researchers, the *Global Workspace Theory* by Bernard Baars[4] is a foundational inpiration.

At the core of the architecture is the concept that *consciouness* can be seen as a *stage* on which many *actors* (also called *processors*) vy for attention via a special kind of weighted messages (called *chunks*) that get propagated from the *actors* all the way to the stage. Because the *stage* has a very limited capacity, only the *chunks* with the highest weights can be accepted at the same time on the *stage*. Once there, all chunks will be broadcasted out to all the *actors* (also called *audience*) at a set pace. At no point in time the actors are said to be *conscious*, even when they have one of their *chunk* on *stage*. Their activity is *unconscious*. Contrasting to the limited nature of the *conscious stage*, the *unconscious* actors are numerous and highly parallelized.

The *chunks* that are broadcasted by the *stage* set the *focus* of the system. While many other unconscious processes may be on-going at any time, *processors* are expected to paid careful attention to what is happening on the *stage* and react accordingly. The main advantage of this architecture, as resoned by Manuel Blum, is to allow for a better handling of complex situations, even unanticipated ones. But, it would also allow for a system to create the most likely interpretation of the *world* in which it operates, and continuously re-evaluate that interpretation and adjust accordingly.

Since Professor Blum and his collaborators have not, at the time of this writing, published an article on the subject, a short description of the main components of a CTM, as well as some key details on their functioning, is needed as we will base our implementation on it. This was gleaned by the author from talks' notes and email exchanges with Professor Blum:

- Chunk (the basic element that is send to/from the *stage*) contains:
    - a GUID (that uniquely identify the *chunk*)

[1]http://f1zz.org/downloads/iex.pdf
[2]http://f1zz.org/downloads/ctm.tgz
[3]https://www.cs.cmu.edu/~mblum/
[4]http://bernardbaars.pbworks.com/w/page/14637775/FrontPage

  - a label (which identify the type/source of the *chunk*)
  - a payload (e.g. a number, symbol ...)
  - a weight (a floating point value)

- Stage (conscious)
  - Limited in capacity (7±2 *chunks*)
  - Only accept *chunks* above a certain threshold (which value can change dynamically)
  - Some *chunk* can kick every other *chunk* of the *stage* (e.g. pain)
  - Regularly broadcasts to the audience the *chunks* with the biggest weights
  - Is not a "central executive" (there may be central executive processors, but their access to the *stage* is handled like any other processors)
  - Has no processing/inferring abilities
  - *Chunk*'s weight decays over time when the *chunk* is *on stage*
  - Very dynamic

- Actor (unconscious)
  - See the *stage*'s *chunk* broadcasts (and react to them if possible/needed)
  - Highly specialized
  - Highly connected to other *actor*s (query/reply)
    - connection between *actors* can be non-direct (via the *stage*) when the *actors* don't know about each others yet (once an *actor* answers to another one via *chunk*, it may gets a link to the the answering *actor*)
    - linking enable *conscious* processing to go *unconscious*
  - Attempt to *go conscious* by the weight of the *chunk* it pushes up
    - weight grows as a function of importance and length of time it has been put off
    - what get to the *stage* may get integrated between many *actors* that are associated. In that case the *chunk* with the highest magnitude is the one getting pushed-up, with the weight as the sum of the weights.
  - An *actor*'s weight:
    - grows over time while it is *on* stage
    - decays over time while there's no interaction with other *actors*
    - changes when the *actor* provides good/bad answers

For a better (and deeper) introduction to the *Conscious Turing Machine*, it is recommended to watch one of the numerous talks that Professor Blum have given over the past few years[5].

The *stage*, *chunks* and *actors*

As discussed in the previous section, a CTM is composed of a *stage* and a (potentially) large set of *actors*, with special messages being exchanged between the *stage* and the *actors*. To implement this in *fizz* , we are going to use a series of *elementals* which will work together to implement the *stage* and similarly use as many *elementals* as necessary to implement any *actors*.

Chunks will be represented as *list* of four *terms*, which will be wrapped into *statements*. We are going to use the *statement*'s label to allow for a more efficient filtering of the *chunks* during runtime. We will also use special labels to indicate events such as the first time a *chunk* is accepted on *stage* as well as when it is dropped from the *stage*. Note that it isn't something that is specified in the Blum's CTM, but it is practical.

[5]for example https://www.youtube.com/watch?v=AXKI2f1AxtM from October 2018

`ctm.chunk.c`	conscious chunk (stage to actors)
`ctm.chunk.u`	unconscious chunk (actor to stage)
`ctm.chunk.d`	chunk drops from the stage
`ctm.chunk.j`	chunk joins into the stage

The expected *terms* in each of these *chunks* will be: two *symbols* (a unique identifier and a label), a *term* and a *number*, the weight of the *chunk*.

To make it easier for an *actor* to be aware of all *chunks* on *stage* at once, we will also broadcast the *chunks* as a (weight) sorted *list* and as a *frame* in which the *chunks* will be grouped by their label:

`ctm.chunk.c.l`	list of all conscious chunks (stage to actors), sorted by increasing weights
`ctm.chunk.c.f`	frame of all conscious chunks (stage to actors)

At regular intervals, the *stage* will go over all the *chunks* it has accepted (in respect to its limited capacity) and broadcast them out. Any *actor* that cares for specific *chunks* is then able to react to them. This is something which we will be using to monitor the activity of the CTM as any *elementals* can listen to any *statements*.

Let's get started now by setting up the ability to broadcast out (to all *actors*) at a regular time interval. To do that, we create a new *fizz* file called `ticks.fizz` and specify in it an instance of the *elemental* class `FZZCTicker` which we will call `ctm.tick.fast`:

```
ctm.tick.fast {
    class           = FZZCTicker,
    tick            = $tick.fast,
    tick.on.attach  = yes
} {}
```

The `tick` property specifies the frequenty (in seconds) at which the *elemental* will *declare* a *statement*. When a *statement* in *fizz* is *declared*, it is broadcasted out to all *elementals* that cares (by having a matching trigger *predicate*) in the *substrate*. Thus, we can use such *statement* to trigger periodic inferrings.

Since not everything that will be running in a CTM is necessary bound to the same frequency, we are going to add two more *tickers*:

```
ctm.tick.dull {
    class           = FZZCTicker,
    tick            = $tick.dull,
    tick.on.attach  = yes
} {}

ctm.tick.slow {
    class           = FZZCTicker,
    tick            = $tick.slow,
    tick.on.attach  = yes
} {}
```

To set the *tick* property of the *tickers*, we have used *constants* (`$tick.slow`, `$tick.dull` and `$tick.fast`). This allow us to treat them as parameters that can easily be changed when an experiment is started. To do that, and make things easier, we are going to use a *solution file* which on top of providing a value for the *constants* will also specify all the *fizz* files we wish to load.

Create a new `JSON` file called `stage.json` and copy into it the following:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["ticks.fizz"],
        "globals" :   [
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            }
        ]
    }
}
```

The file indicates that `ticks.fizz` should be loaded and that three constants should be registered by the runtime, with value going from `0.5` to `3` seconds. We will use the `ctm.tick.fast` ticker for the *stage* broadcasts. A more realistic[6] value here than `0.5` would be `0.25` but having a slower pace make testing and developing a bit easier, so we will keep it at that value for now.

Let's give what we have so far a try, using the console *command* `spy` to verify that indeed the *elemental* `ctm.tick.slow` is *declaring* a *statement* every seconds. Note that instead of loading a *fizz* file, we are given as command line argument the JSON file we just created. As long as the file conforms to what is expected, *fizz* will *interprete* it:

```
$ ./fizz.x64 ./etc/experiments/ctm/stage.json
fizz 0.5.D-X (20181108.1239) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/stage.json ...
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading completed in 0.004s
?- /spy(append,ctm.tick.slow)
spy : observing ctm.tick.slow
spy : S ctm.tick.slow(9, 1541794413.795849) (90.000000)
spy : S ctm.tick.slow(10, 1541794414.796397) (90.000000)
spy : S ctm.tick.slow(11, 1541794415.796729) (90.000000)
spy : S ctm.tick.slow(12, 1541794416.796212) (90.000000)
spy : S ctm.tick.slow(13, 1541794417.796727) (90.000000)
spy : S ctm.tick.slow(14, 1541794418.796333) (90.000000)
```

Without a way to see what's happening when a CTM is running, it is going to be difficult to test, debug or verify that it is working as expected. To help with that we are now going to setup an *elemental* which sole purpose will be to output to the console any *chunks* that get declared. Create a new *fizz* file called `debug.fizz` in which we will be creating a *prototype* (a.k.a. a *rule*) with a *trigger predicate* for each of the *chunk*'s label we defined earlier:

```
ctm.chunk.observer {

    () :-   @ctm.chunk.c.l(:l?[neq([])]),
            console.puts("ctm.obs: ",ctm.chunk.c.l(:l)),
            hush;

    () :-   @ctm.chunk.c.f(:f), frm.length(:f,_?[gt(0)]),
            console.puts("ctm.obs: ",ctm.chunk.c.f(:f)),
            hush;

    () :-   @ctm.chunk.c(:g,:s,:d,:w),
```

[6] https://www.princeton.edu/news/2018/08/22/spotlight-attention-more-strobe-say-researchers

```
            console.puts("ctm.obs: ",ctm.chunk.c(:g,:s,:d,:w)),
            hush;

    () :-   @ctm.chunk.u(:g,:s,:d,:w),
            console.puts("ctm.obs: ",ctm.chunk.u(:g,:s,:d,:w)),
            hush;

    () :-   @ctm.chunk.d(:g,:s,:d,:w),
            console.puts("ctm.obs: ",ctm.chunk.d(:g,:s,:d,:w)),
            hush;

    () :-   @ctm.chunk.j(:g,:s,:d,:w),
            console.puts("ctm.obs: ",ctm.chunk.j(:g,:s,:d,:w)),
            hush;

    () :-   @ctm.chunk.r(:g,:s,:d,:w),
            console.puts("ctm.obs: ",ctm.chunk.r(:g,:s,:d,:w)),
            hush;

}
```

When, for example, a *chunk* is sent by an *actor* (using `ctm.chunk.u`) we will unify four *variables* to its *terms* and print them in the console. Note that call to the `hush` primitive which end each of the *prototype* isn't necessary for this to work, but is present here as a minor performance optimization as it prevent the *elemental* from itself *declaring* a *statement* on successful resolution of any *prototype*. We will be using this throughout this article.

Once we add the file `debug.fizz` to the *solution file* `stage.json`, we can test `ctm.chunk.observer` as follow:

```
$ ./fizz.x64 ./etc/experiments/ctm/stage.json
fizz 0.5.D-X (20181108.1239) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/stage.json ...
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loaded ./etc/experiments/ctm/debug.fizz in 0.009s
load : loading completed in 0.011s
?- declare(ctm.chunk.u(%sym.4,hello,[1,2,3,4],1))
-> (  ) := 1.00 (0.001) 1
ctm.obs: ctm.chunk.u(vbwh, hello, [1, 2, 3, 4], 1)
```

Here, we have used the `declare` *primitive* to *broadcast* into the *substrate* a *statement* build from the `ctm.chunk.u` *functor* (with an arity of 4) that is passed as *term* to the *primitive*. Note that `%sym.4` is a *volatile* which get substitued by a random *symbol* with a length of four characters. We will use that intensively in this article to generate *GUID* since we will not be building anything where a large number of *chunks* may lead to *GUID* collisions. For a real-world scenario, it will be preferable to use `%sym.10` or `%sym`.

Let's now look at implementing the *stage* it-self. We will put all *elemental* definitions in a single *fizz* file called `stage.fizz` which we will need to add to the list of files to be loaded in `stage.json`. If you recall the CTM description from earlier, the role of the *stage* is to see all incoming *chunks* and having kept only the ones with the highest weight values (since the *conscious* capacity is limited by the *Magical Number Seven, Plus or Minus Two*[7]), broadcasts them all out to whomever cares. There, is one of the difference between the CTM devised by *The Blums* and this experimental implementation as they only see the broadcasting to include a single *chunk*, the one with the highest weight, and we broadcast out every single *chunks* on stage.

The last thing we need the *stage* to do is to lower the weight of the *chunks* over time so that they gets removed once the value drops below a given threshold (that is unless the *chunk* get replaced by a newer version of it-self with a higher weight value).

[7]https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

Since any *chunks* may arrive at anytime, while we are only broadcasting the *chunks* on *stage* every so often, we are going to store them temporary in a *frame* using the *chunk*'s `GUID` as the key. This will allow us to always keep the most recent version of a given *chunk* if multiple versions of it are sent in shorter succession than the time in between *stage*'s broadcast.

So, to get the main *elemental* for the *stage* started, we will define its properties as follow:

```
ctm.stage {

    chunks = {},
    c.list = {},
    c.size = $stage.size,
    c.loss = $stage.loss,
    c.drop = $stage.drop

} {}
```

The `chunks` property will be a *frame* containing all the chunks that are currently *on the stage* and `c.list`, a *frame*, will hold the recently received *chunks* temporary until the *elemental* get to a broadcasting cycle. With the following three properties, we respectively indicates the maximum number of *chunks* accepted *on stage*, the weight loss to be applied to each *chunk* and lastly the threshold weight under which a *chunk* will get removed from the *stage*.

As we did for the tickers' tick we defined earlier, we are using *constants* to set these three last properties. So we need to edit the `stage.json` file and add them:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["ticks.fizz", "debug.fizz", "stage.fizz"],
        "globals" :   [
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            },
            {
                "label" : "stage.size",
                "value" : 7
            },
            {
                "label" : "stage.loss",
                "value" : 0.1
            },
            {
                "label" : "stage.drop",
                "value" : -2
            }
        ]
    }
}
```

With the *stage* properties now in place, we can now add to the *elemental* the *prototypes* that will be triggered when we get a tick and when we get any *unconscious chunks*:

```
ctm.stage {

    chunks = {},
 4      c.list = {},
 5      c.size = $stage.size,
 6      c.loss = $stage.loss,
 7      c.drop = $stage.drop
 8
 9  } {
10
11      // broadcast all the chunks on the stage (after purging them)
12      ()  :-  @ctm.tick.slow(_,:t),
13              peek(c.list,:c.wait),
14              poke(c.list,{}),
15              frm.values(:c.wait,:c.list), frm.values($chunks,:chunks),
16              #ctm.stage.chunks.update(:chunks,$c.loss,$c.drop,:f),
17              #ctm.stage.chunks.inject(:c.list,:t,:f,:f.r),
18              #ctm.stage.chunks.resize(:f.r,:t,$c.size,:v.o),
19              #ctm.stage.list.to.chunks(:v.o,{},:f.o),
20              #ctm.stage.list.to.public(:v.o,:v),
21              poke(chunks,:f.o),
22              #ctm.stage.chunks.broadcast(:v.o,:t),
23              #ctm.stage.list.to.frame(:v.o,{},:f.c),
24              declare(ctm.chunk.c.l(:v)),
25              declare(ctm.chunk.c.f(:f.c)),
26              hush;
27
28      // handle chunk.u, we append it to a temporary storage
29      ()  :-  @ctm.chunk.u(:g?[is.bound],:s?[is.bound],:d?[is.bound],:w?[is.bound]),
30              frm.store($c.list,:g,[:g,:s,:d,:w],:f.o),
31              poke(c.list,:f.o),
32              hush;
33
34  }
```

The *prototype* on line 29, starts by specifying a *trigger predicate* for the *chunks* sent to the *stage*. For sanity sake, it checks that the four *terms* are bound (that is no unbound *variables*) before unifying their values to four local *variables*. When that is satisfied, the *prototype* will store a *list* that contains all four *terms* in the property `c.list` using the *chunk*'s *GUID* as the key. As in *fizz* all *terms* as immutable, the primitive `frm.store` which we use here, will unify it's last *term* with the new *frame* that results from storing a new value in the *frame*. The *prototype* ends by using the `poke` primitive to write the new *frame* (bound to the variable `f.o`) in the *elemental* property `c.list`.

The first *prototype* which starts on line 12, get executed for each *statements* generated by one of the tickers (we use *wildcard variables* for the unification of its first *term* as we will only care about the second one, the timestamp). It starts by reading the *frame* stored in the property `c.list` and right away replace it (in the *elemental*'s properties) by an empty *frame*. This is done by using the primitives `peek` and `poke`. The next step taken by the *prototype* is to extract all the chunks stored in the *frame* with the primitive `frm.values`. This will unify the *variable* `:c.list` with a list of all the *chunks* received since the last tick. The inferring continues with *predicates* calling on five different *secondary elementals*. It updates the weight (substract from each the amount taken from the property `c.loss`) of all the chunks currently on stage (using `ctm.stage.chunks.update`) as well as drop any *chunk* which weight is below a given threshold (given by the property `c.drop`). We then insert into the *frame* that contains all the updated *chunks* the new ones (with `ctm.stage.chunks.inject`). The second *term* of that *predicate* is the timestamp from the ticker. We use it to keep track of when a *chunk* was first added to the *stage*. The *elemental* `ctm.stage.chunks.resize` is used to insure that the *frame* won't have more items than the *stage* can accept (using the property `c.size`). The *variable* `v.o` will be unified with a *list* that contains all the *chunks* accepted on stage. We will then relies on `ctm.stage.list.to.chunks` to convert the *list* into a *frame* then use `ctm.stage.list.to.public` to convert the same *list* into a *list* where the timestamp has been removed from all *chunks*. Finally we use the *primitive* `poke` to write the *frame* back into the *elemental*'s properties.

The line 22 to 25 completes the handling of the ticker's tick by broadcasting out to the *actors* all the *chunks* on *stage* (using `ctm.stage.chunks.broadcast`), then we convert the *list* in a *frame* were all the *chunks* are grouped by labels (`ctm.stage.list.to.frame`) and finally declare both the *list* and the *frame*.

Let's now look at the first secondary *elemental* which we used in `ctm.stage`:

```
 1 ctm.stage.chunks.update {
 2
 3     ([],_,_,{})^                                :-  true;
 4
 5     ([[:g,:s,:d,:w,:t]],:loss,:drop,:f.o)       :-  sub(:w,:loss,:w2), gt(:w2,:drop)^,
 6                                                     frm.store({},:g,[:g,:s,:d,:w2,:t],:f.o);
 7     ([[:g,:s,:d,:w,:t]],:loss,:drop,{})         :-  sub(:w,:loss,:w2), lte(:w2,:drop)^,
 8                                                     #ctm.stage.callback(d,[:g,:s,:d,:w2]);
 9     ([[:g,:s,:d,:w,:t]|:r],:loss,:drop,:f.o)    :-  sub(:w,:loss,:w2), gt(:w2,:drop)^,
10                                                     frm.store({},:g,[:g,:s,:d,:w2,:t],:f.i2),
11                                                     ~self(:r,:loss,:drop,:f.m),
12                                                     frm.cat(:f.i2,:f.m,:f.o);
13     ([[:g,:s,:d,:w,:t]|:r],:loss,:drop,:f.o)    :-  sub(:w,:loss,:w2), lte(:w2,:drop)^,
14                                                     #ctm.stage.callback(d,[:g,:s,:d,:w2]),
15                                                     ~self(:r,:loss,:drop,:f.o);
16
17 }
```

The *elemental* contains five *prototypes* whose entrypoints matchs the followings: the *list* of *chunks* to update, the weight loss and weight drop values. The last *term* will be the *frame* which will contains the updated *chunks* (except all the ones that were removed). Since the *elemental* will be traversing a *list*, its *procedural knowledge* follow a classic recursive pattern, where a *prototype* deals with an empty *list* (line 3), one deals with a *list* with a single item in it (line 5 and 7), and finaly one *prototype* deals with a *list* of two or more items (line 9 and 13). The gotcha here is that since not all the *chunks* on the list are going to be included in the result *list*, each of the *prototypes* that deals with a *chunk* need to be duplicated: one for when the reduced weight is above the drop value and one for when the weight is below the threshold. When the *chunk* isn't added to the *frame*, we will call `ctm.stage.callback` to broadcast the fact that a *chunk* was dropped from the stage.

Let's briefly unpack some of the *procedural knowledge* we just wrote. The *caret* character is used in *fizz* to indicate that backtracking and concurrent inferring must be disabled once the *predicate* to its left has succeeded. When it is used with a *prototype*'s entrypoint, no other *prototype* from the *elemental* will be considered when the entrypoint unifies with a *query*'s *predicate*. The last two *prototypes* recursively call the *elemental* using the `~self` syntax. Using the `self` keyword not only make writing recursive *predicates* simpler, it also allow for an *elemental* to be cloned and have the *predicate* points to the right *elemental*.

The next *secondary elemental* we are going to look at is `ctm.stage.chunks.inject`. As we have briefly stated earlier, its purpose is to insert all the *chunks* we have received (stored in a *list*) into a *frame* containing all the *chunks* still on the *stage*. This is done in a way that is similar to the previous *elemental* we looked at. Since we are using a *frame* (with the *chunk*'s GUID) as the label, replacing a given *chunk* by a new instance of it would be easy if it wasn't for the fact that we are also storing the timestamp at which a *chunk* got into the *stage*. When we are replacing it, we want the timestamp to be conserved:

```
 1 ctm.stage.chunks.inject {
 2
 3     ([],_,:f.i,:f.i)^                   :-  true;
 4
 5     ([[:g,:s,:d,:w]],:t,:f.i,:f.o)      :-   frm.fetch(:f.i,:g,[_,_,_,_,:t0])^, frm.store(:f.i,:g,[:g,:s,:d,:w,:t0],:f.o);
 6     ([[:g,:s,:d,:w]],:t,:f.i,:f.o)      :-  !frm.fetch(:f.i,:g,[_,_,_,_,:t0])^, frm.store(:f.i,:g,[:g,:s,:d,:w,:t],:f.o);
 7
 8     ([[:g,:s,:d,:w]|:r],:t,:f.i,:f.o)   :-   frm.fetch(:f.i,:g,[_,_,_,_,:t0])^, frm.store(:f.i,:g,[:g,:s,:d,:w,:t0],:f.i2),
 9                                             ~self(:r,:t,:f.i2,:f.o);
10     ([[:g,:s,:d,:w]|:r],:t,:f.i,:f.o)   :-  !frm.fetch(:f.i,:g,[_,_,_,_,:t0])^, frm.store(:f.i,:g,[:g,:s,:d,:w,:t],:f.i2),
11                                             ~self(:r,:t,:f.i2,:f.o);
12
13 }
```

On line 3, we defined a *prototype* for dealing with an empty *list*. It is in *fizz* necessary to have such statement in our case as otherwise, when the *stage* is getting re-evaluated with no new *chunks* to be considered, none of the *prototypes* in the *elemental* will be used. This will lead to the *predicate* in `ctm.stage` to not be answered since no other *elemental* can handle this *predicate*. Eventually, the query from `ctm.stage` will get discarded.

Line 5 and 6 deals with adding a single (or the last) *chunk*; We either succeeded fetching the *chunk* using its GUID when the *chunk* already exists (using the *primitive* `frm.fetch`) so that we can keep the timestamp (which value will be bound to the *variable* `t0`) and use it for the new *list* that will contains the *chunk* stored in a new *frame*. If `frm.fetch` fails (on line 6), the negation indicator (`!`) will allow the inferring to continue so that we store the new *chunk* using the timestamp bound to the *variable* `t`. Line 8 and 10 shows the same pattern, but also deals with traversing the *list* by recursively calling the *elemental* on the rest (bound to the *variable* `r`) of the *list* except the head (the first element).

The *secondary elemental* that follows is `ctm.stage.chunks.resize`, which takes the *frame* containing the updated and the new *chunks* and insure that we respect the *stage* capacity, by removing as many (lowest weight first) *chunks* as needed:

```
 1 ctm.stage.chunks.resize {
 2
 3     (:f.i,:t,:n,:v.s)  :-   frm.length(:f.i,:l), lte(:l,:n)^,
 4                             frm.values(:f.i,:v.i), lst.sort(:v.i,:v.s,3);
 5     (:f.i,:t,:n,:v.o)  :-   frm.length(:f.i,:l), frm.values(:f.i,:v.i),
 6                             lst.sort(:v.i,:v.s,3), sub(:l,:n,:d),
 7                             lst.tails(:v.s,:n,:v.o), lst.heads(:v.s,:d,:v.d),
 8                             #ctm.stage.chunks.broadcast.drops(:v.d,:t);
 9
10 }
```

In *fizz* , as you may have guessed, the order in which the *prototypes* are defined matters when they are considered for answering a query. The first *prototype* on line 3, uses the *primitive* `frm.length` to get the number of *chunks* in the *frame* and compares that to the *stage* capacity that is given as the third *term* of the *predicate* that unifies with the *prototype*'s entrypoint (*variable* `n`). If the number of *chunks* is less or equal to the capacity, the inferring will continue by getting a list of all the chunks in the *frame* (using `frm.values`) then sorting the *list* using the primitive `lst.sort`. The third *term* in the `lst.sort` *predicate* is the index of the *term* in each of the sub-*list* contained in the *list* as the value to sort on (increased order). Since we have used the *cut* indicator to the right of the `lte` *predicate*, none of the other *prototypes* will be considered. Thus, the *prototype* on line 5, doesn't need to check that the number of items in the input *frame* is larger than the *stage* capacity. Just as the previous *prototype*, the inferring get the list of *chunks* and sort it before splitting the *list* into two parts, the one that will be keept (bound to the *variable* `v.o`) and the one that will be dropped (bound to `v.d`). The inferring then finish by calling the *elemental* `ctm.stage.chunks.broadcast.drops` to broadcast all the *chunks* that have been dropped from the *stage*.

Now that `ctm.stage` has the *list* of the *chunks* on *stage* when it gets the reply to the `ctm.stage.chunks.resize` *predicate*, it queries `ctm.stage.list.to.chunks` to transform the *list* into a *frame*:

```
ctm.stage.list.to.chunks {

    ([],:f,:f)^                           :- true;
    ([[:g,:s,:d,:w,:t]],:f.i,:f.o)^       :- frm.store(:f.i,:g,[:g,:s,:d,:w,:t],:f.o);
    ([[:g,:s,:d,:w,:t]|:r],:f.i,:f.o)     :- frm.store(:f.i,:g,[:g,:s,:d,:w,:t],:f.r), ~self(:r,:f.r,:f.o);

}
```

Here again, the pattern used for the *procedural knowledge* is the same as we are traversing a *list* to recursively build a *frame*. That same pattern is again used for the *secondary elemental* used by `ctm.stage` to convert the internal *list* of *chunks* into one that we can send out to any *actors* interested about the content of the *stage* as a whole:

```
ctm.stage.list.to.public {

    ([],[])^                                      :- true;
    ([[:g,:s,:d,:w,_]],[[:g,:s,:d,:w]])^          :- true;
    ([[:g,:s,:d,:w,_]|:r],[[:g,:s,:d,:w]|:r2])    :- ~self(:r,:r2);

}
```

This is however, a simpler *procedural knowledge* which rebuild the *list* as given in the first *term* of the *predicate* minus the fifth element of the *list* representing a *chunk*. Once the query from `ctm.stage` is answered by the *elemental*, the inferring resumes, moving to broadcasting the content of the *stage* to the actors. This is done by the *elemental* `ctm.stage.chunks.broadcast` which is defined as:

```
ctm.stage.chunks.broadcast {

    ([],_)^                        :-  true;
    ([[:g,:s,:d,:w,:t]],:t)^       :-  #ctm.stage.callback(j,[:g,:s,:d,:w]);
    ([[:g,:s,:d,:w,_]],_)^         :-  #ctm.stage.callback(c,[:g,:s,:d,:w]);
    ([[:g,:s,:d,:w,:t]|:r],:t)^ :-  #ctm.stage.callback(j,[:g,:s,:d,:w]), ~self(:r,:t);
    ([[:g,:s,:d,:w,_]|:r],:t)^  :-  #ctm.stage.callback(c,[:g,:s,:d,:w]), ~self(:r,:t);

}
```

The *elemental* expects the first *term* to any predicate targeting the *elemental* to be the public *list* of *chunks* we just computed above, and the timestamp of the current *stage* evaluation. This value is used to decide which of the broadcasting labels to use for any given *chunk*. When the timestamp value matches the one in the *chunk*, we want the stage to broadcast the *chunk* as `ctm.chunk.j`, indicating that the *chunk* is new to the *stage*. When the timestamps don't match, it is then `ctm.chunk.c` that should be used.

The last *secondary elemental* directly used by `ctm.stage` is `ctm.stage.list.to.frame`. It takes the *list* of *chunks* and create a *frame* from it where all *chunks* with the same label are grouped together:

```
1   ctm.stage.list.to.frame {
2
3       ([],:f,:f)^                            :-   true;
4       ([[:g,:s,:d,:w,_]],:f.i,:f.o)          :-  !frm.label(:f.i,:s)^, frm.store(:f.i,:s,[[:g,:d,:w]],:f.o);
5       ([[:g,:s,:d,:w,_]],:f.i,:f.o)          :-   frm.label(:f.i,:s)^, frm.fetch(:f.i,:s,:l),
6                                                   frm.store(:f.i,:s,[[:g,:d,:w]|:l],:f.o);
7       ([[:g,:s,:d,:w,_]|:r],:f.i,:f.o)       :-  ~self(:r,:f.i,:f.r), !frm.label(:f.r,:s)^, frm.store(:f.r,:s,[[:g,:d,:w]],:f.o);
8       ([[:g,:s,:d,:w,_]|:r],:f.i,:f.o)       :-  ~self(:r,:f.i,:f.r), frm.label(:f.r,:s)^, frm.fetch(:f.r,:s,:l),
9                                                   frm.store(:f.r,:s,[[:g,:d,:w]|:l],:f.o);
10  }
```

The way this *procedural knowledge* works here is similar to `ctm.stage.chunks.inject`: Line 4 and 5 deals with the last (or only) *chunk* in the *list*. Line 4 creates the pair in the *frame* for the label of the *chunk* while line 5, add to an existing pair in the *frame*. Line 7 and 8 do the same thing, with the added need to recursively work on the rest of the *list*.

We now need to quickly look at the two support *elementals* we have used, but not yet defined: `ctm.stage.chunks.broadcast.drops` and `ctm.stage.callback`. The first one, as you may recall, is called upon by `ctm.stage.chunks.resize` to inform the *actors* of any *chunk* that have been dropped from the *stage* as having the lowest weight in conditions where the *stage* is above capacity. Its *procedural knowledge* is defined as:

```
ctm.stage.chunks.broadcast.drops {

    ([],_)^                               :-  true;
    ([[:g,:s,:d,:w,:t]],_?[neq(:t)])^     :-  #ctm.stage.callback(d,[:g,:s,:d,:w]);
    ([[:g,:s,:d,:w,:t]],:t)^              :-  true;
    ([[:g,:s,:d,:w,:t]|:r],:t)^           :-  ~self(:r,:t);
    ([[:g,:s,:d,:w,_]|:r],:t)             :-  #ctm.stage.callback(d,[:g,:s,:d,:w]), ~self(:r,:t);

}
```

The second *term* expected from a *predicate* querying this *elemental* is the timestamp. We use it to differentiate between the *chunks* that have just been added to the *stage* and thoses that have been on *stage* for at least one tick. When such *chunk* is to be dropped, we simply won't broadcast that as the initial broadcast upon joining the *stage* hasn't been send. Line 4 makes use of a *constrained wildcard* as its second *term* to insure that any *predicate* unifying with the entrypoint and thus executing a call to `ctm.stage.callback` will be for a *chunk* that was not just added to the *stage*.

Let's now continue and look at *ctm.stage.callback*:

```
ctm.stage.callback {

    (d,[:g,:s,:d,:w]) :- declare(ctm.chunk.d(:g,:s,:d,:w));
    (j,[:g,:s,:d,:w]) :- declare(ctm.chunk.j(:g,:s,:d,:w));
    (c,[:g,:s,:d,:w]) :- declare(ctm.chunk.c(:g,:s,:d,:w));

}
```

It contains three *prototypes*, one for each of the supported broadcasts and make use of the *primitive* `declare` to broadcast out a *statement* for a *chunk*. The necessity of this *elemental* can be questioned since it only calls upon a single *primitive* in a straightforward fashion. We are, however, calling it from many different places, so if we were to change or add to the way a *chunk* is broadcasted to the *actors*, having the possiblity of changing it in a single location is advantageous.

We now have a complete set of *elementals* that implement the basic working of the *stage*. Before moving on to complexe examples of CTM in action, let's test that the *stage* is working as we would expect with a couple of simple examples.

The first example is going to use keypress events as *chunks* being sent by an *actor* directly to the *stage*. To start, create a JSON file called `key.json` with the following content in it:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["stage.fizz", "debug.fizz", "ticks.fizz", "key.fizz" ],
        "globals" :   [
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            },
            {
                "label" : "stage.size",
                "value" : 7
            },
            {
                "label" : "stage.loss",
                "value" : 0.1
            },
            {
                "label" : "stage.drop",
                "value" : -0.1
            }
        ]
    }
}
```

It is basically an adaptation to the JSON file we setup earlier (`stage.json`), adding a new *fizz* file called `key.fizz` to the list of the file to be loaded. In that new *fizz* file we are going to create our first *actor*, which when the user presses a key on the keyboard will attempt to push a *chunk* based on that event onto the *stage*. Here's the definition of that actor, which we will call `ctm.actor.keypress`:

```
 1 ctm.actor.keypress {
 2
 3     weight.value = 1,
 4     weight.range = <0.2|1>,
 5     weight.loss  = 0.1,
 6     started      = no
 7
 8 } {
 9
10     // send a chunk for every keypress we get
11     ()  :-  @console.keypress(:k),
12             ~self(uid,:g),
13             declare(ctm.chunk.u(:g,keypress,:k,$weight.value)),
14             poke(started,yes),
15             hush;
16
17     // lower the weight as time flow
18     ()  :-  @ctm.tick.fast(_,_),
19             peek(started,yes),
20             sub($weight.value,$weight.loss,:w.o),
21             rng.clamp($weight.range,:w.o,:w.c),
22             poke(weight.value,:w.c),
23             hush;
24
25     // each time the keypress chunk is on stage we increase the weight
26     ()  :-  @ctm.chunk.c(_,keypress,_,_),
27             add($weight.value,$weight.loss,:w.o),
28             rng.clamp($weight.range,:w.o,:w.c),
29             poke(weight.value,:w.c),
30             hush;
31
32     // get randomly assigned uid or create one if needed
33     (uid,:u) :- peek(uid,:u)^;
34     (uid,:u) :- set(:u,%sym.4), poke(uid,:u);
35
36 }
```

The *procedural knowledge* it contained is composed of three *prototypes* each called into inferring by a *trigger predicate*. The first one, on line 11, gets going when the user press a key on the keyboard (which cause a `console.keypress` *statement* to be broadcasted in the *substrate*) and after calling the *elemental* itself to get a GUID (assigned the very first time a key is pressed, see line 33 and 34), uses the `declare` *primitive* to send a *chunk* to the stage. The inferring then end by setting the value of the `started` property of the *elemental* to `yes`. This property is queried by the second *prototype* (on line 19) to prevent the weight of the *chunks* emitted by the *actor* from getting reduced at each tick before the user have started interacting with the *actor*. After the first *chunk* is emitted by the *actor*, the weight of the next *chunk* will over time decrease by substracting from the property `weight.value` the value in `weight.loss` (using the *primitive* `sub`). The *prototype* also clamps `weight.value` using the *range* indicated in its property `weight.range`. The third *prototype* on line 26 get triggered when one of the *chunk* on *stage* (the *predicate*'s label is `ctm.chunk.c`) is the *chunk*. When this occurs, the inferring will increase the weight of the next *chunk* in a way that mirror what we did in the second *prototype*.

Let's now try this. Note, though, that the actor always send the same *chunk* to the *stage* as the GUID used for it is always the same, even if the actual keycode changes:

```
$ ./fizz.x64 ./etc/experiments/ctm/key.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/key.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.008s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/key.fizz ...
load : loaded ./etc/experiments/ctm/key.fizz in 0.005s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.037s
load : loading completed in 0.040s
ctm.obs: ctm.chunk.u(wpya, keypress, 113, 1)
ctm.obs: ctm.chunk.j(wpya, keypress, 113, 1)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.900000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.800000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.700000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.600000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.500000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.400000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.300000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.200000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0.100000)
ctm.obs: ctm.chunk.c(wpya, keypress, 113, 0)
ctm.obs: ctm.chunk.d(wpya, keypress, 113, -0.100000)
```

Once the loading of the JSON file is completed, the *stage* is up and running. We then pressed once the `Q` key which caused the *statement* `ctm.chunk.u(wpya, keypress, 113, 1)` to be declared and picked-up by the *stage* for inclusion on the *stage*. As expected, a `ctm.chunk.j` *statement* is declared and then followed every half-seconds by `ctm.chunk.c` *statements* with a decreasing weight. Eventually, the weight of the *chunk* which haven't been declared again by the *actor* drops to the threshold and gets removed from the *stage* causing a `ctm.chunk.d` *statement* to be declared.

Let see how the system handles the user pressing different keys:

```
ctm.obs: ctm.chunk.u(pydq, keypress, 113, 1)
ctm.obs: ctm.chunk.j(pydq, keypress, 113, 1)
ctm.obs: ctm.chunk.c(pydq, keypress, 113, 0.900000)
ctm.obs: ctm.chunk.u(pydq, keypress, 119, 0.800000)
ctm.obs: ctm.chunk.c(pydq, keypress, 119, 0.800000)
ctm.obs: ctm.chunk.u(pydq, keypress, 101, 0.600000)
ctm.obs: ctm.chunk.c(pydq, keypress, 101, 0.600000)
ctm.obs: ctm.chunk.c(pydq, keypress, 101, 0.500000)
ctm.obs: ctm.chunk.u(pydq, keypress, 116, 0.400000)
ctm.obs: ctm.chunk.c(pydq, keypress, 116, 0.400000)
ctm.obs: ctm.chunk.c(pydq, keypress, 116, 0.300000)
ctm.obs: ctm.chunk.c(pydq, keypress, 116, 0.200000)
ctm.obs: ctm.chunk.c(pydq, keypress, 116, 0.100000)
ctm.obs: ctm.chunk.c(pydq, keypress, 116, 0)
ctm.obs: ctm.chunk.d(pydq, keypress, 116, -0.100000)
```

We can see that the *stage* replaced the *chunk* that was on stage by the newer version, using the weight that was set in the `ctm.chunk.u` *statement*. As some time passed before the two keypresses, the weight used for the *statement* by the *actor* isn't `1`. Further keypresses shows the same effect.

By commenting the right *prototype* in `debug.fizz` and uncommenting the one using `ctm.chunk.c.l` as *trigger predicate*, we can observe the full content of the *stage* in a single debug output:

```
ctm.obs: ctm.chunk.u(wqoc, keypress, 97, 1)
ctm.obs: ctm.chunk.j(wqoc, keypress, 97, 1)
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 97, 1]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 97, 0.900000]])
ctm.obs: ctm.chunk.u(wqoc, keypress, 122, 0.600000)
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 122, 0.600000]])
ctm.obs: ctm.chunk.u(wqoc, keypress, 120, 0.500000)
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0.500000]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0.400000]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0.300000]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0.200000]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0.100000]])
ctm.obs: ctm.chunk.c.l([[wqoc, keypress, 120, 0]])
ctm.obs: ctm.chunk.d(wqoc, keypress, 120, -0.100000)
```

To conclude this section, let's look at an evolution of the above example, where each keypress is a different *chunk*. As such, keypress events will *fight* for inclusion on the *stage* as their numbers mount. To start, make a copy of the `key.json` file we created, renaming it `keys.json` and replace in it `key.fizz` by `keys.fizz`. After that, create a new *fizz* file called `keys.fizz` then copy paste into it the following *elemental* definition:

```
ctm.actor.keypress {

    weight.value = 1,
    weight.range = <0.2|1>,
    weight.loss  = 0.1,
    started = no

} {

    // send a chunk for every keypress we get
    ()  :-  @console.keypress(:k),
            declare(ctm.chunk.u(%sym.4,keypress,:k,$weight.value)),
            poke(started,yes),
            hush;

    // lower the weight as time flow
    ()  :-  @ctm.tick.fast(_,_),
            peek(started,yes),
            sub($weight.value,$weight.loss,:w.o),
            rng.clamp($weight.range,:w.o,:w.c),
            poke(weight.value,:w.c),
            hush;

    // each time any keypress chunk is on stage we increase the weight
    ()  :-  @ctm.chunk.c(_,keypress,_,_),
            add($weight.value,$weight.loss,:w.o),
            rng.clamp($weight.range,:w.o,:w.c),
            poke(weight.value,:w.c),
            hush;

}
```

The only difference from this version of the *actor* we created earlier is that we no longer used the same GUID for each of the *chunk* it will emit but instead use the *volatile* `sym.4` to generate a new GUID on the fly. Keeping the changes we made to `debug.fizz`, we can load the new solution file and observe what happens when the user presses `Q W E R T Y` in quick succession:

```
$ ./fizz.x64 ./etc/experiments/ctm/keys.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/keys.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.014s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/keys.fizz ...
load : loaded ./etc/experiments/ctm/keys.fizz in 0.009s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.042s
load : loading completed in 0.045s
ctm.obs: ctm.chunk.u(hyqo, keypress, 113, 1)
ctm.obs: ctm.chunk.j(hyqo, keypress, 113, 1)
ctm.obs: ctm.chunk.c.l([[hyqo, keypress, 113, 1]])
ctm.obs: ctm.chunk.u(gkjx, keypress, 119, 0.800000)
ctm.obs: ctm.chunk.j(gkjx, keypress, 119, 0.800000)
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.800000], [hyqo, keypress, 113, 0.900000]])
ctm.obs: ctm.chunk.u(aidc, keypress, 101, 0.800000)
ctm.obs: ctm.chunk.u(ubgg, keypress, 114, 0.700000)
ctm.obs: ctm.chunk.j(ubgg, keypress, 114, 0.700000)
ctm.obs: ctm.chunk.j(aidc, keypress, 101, 0.800000)
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.700000], [ubgg, keypress, 114, 0.700000], [hyqo, keypress, 113, 0.800000], [
     aidc, keypress, 101, 0.800000]])
ctm.obs: ctm.chunk.u(aktd, keypress, 116, 0.800000)
ctm.obs: ctm.chunk.u(suof, keypress, 121, 0.800000)
ctm.obs: ctm.chunk.j(aktd, keypress, 116, 0.800000)
ctm.obs: ctm.chunk.j(suof, keypress, 121, 0.800000)
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.600000], [ubgg, keypress, 114, 0.600000], [hyqo, keypress, 113, 0.700000], [
     aidc, keypress, 101, 0.700000], [aktd, keypress, 116, 0.800000], [suof, keypress, 121, 0.800000]])
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.500000], [ubgg, keypress, 114, 0.500000], [hyqo, keypress, 113, 0.600000], [
     aidc, keypress, 101, 0.600000], [aktd, keypress, 116, 0.700000], [suof, keypress, 121, 0.700000]])
```

```
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.400000], [ubgg, keypress, 114, 0.400000], [hyqo, keypress, 113, 0.500000], [
     aidc, keypress, 101, 0.500000], [aktd, keypress, 116, 0.600000], [suof, keypress, 121, 0.600000]])
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.300000], [ubgg, keypress, 114, 0.300000], [hyqo, keypress, 113, 0.400000], [
     aidc, keypress, 101, 0.400000], [aktd, keypress, 116, 0.500000], [suof, keypress, 121, 0.500000]])
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.200000], [ubgg, keypress, 114, 0.200000], [hyqo, keypress, 113, 0.300000], [
     aidc, keypress, 101, 0.300000], [aktd, keypress, 116, 0.400000], [suof, keypress, 121, 0.400000]])
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0.100000], [ubgg, keypress, 114, 0.100000], [hyqo, keypress, 113, 0.200000], [
     aidc, keypress, 101, 0.200000], [aktd, keypress, 116, 0.300000], [suof, keypress, 121, 0.300000]])
ctm.obs: ctm.chunk.c.l([[gkjx, keypress, 119, 0], [ubgg, keypress, 114, 0], [hyqo, keypress, 113, 0.100000], [aidc, keypress,
     101, 0.100000], [aktd, keypress, 116, 0.200000], [suof, keypress, 121, 0.200000]])
ctm.obs: ctm.chunk.d(gkjx, keypress, 119, -0.100000)
ctm.obs: ctm.chunk.d(ubgg, keypress, 114, -0.100000)
ctm.obs: ctm.chunk.c.l([[hyqo, keypress, 113, 0], [aidc, keypress, 101, 0], [aktd, keypress, 116, 0.100000], [suof, keypress,
     121, 0.100000]])
ctm.obs: ctm.chunk.d(hyqo, keypress, 113, -0.100000)
ctm.obs: ctm.chunk.d(aidc, keypress, 101, -0.100000)
ctm.obs: ctm.chunk.c.l([[aktd, keypress, 116, 0], [suof, keypress, 121, 0]])
ctm.obs: ctm.chunk.d(aktd, keypress, 116, -0.100000)
ctm.obs: ctm.chunk.d(suof, keypress, 121, -0.100000)
```

As expected, all six *chunks* are getting on *stage* then getting off the *stage* as their weight drops below the threshold. Let see what happens if we press ten keys (key `1` to `0`):

```
ctm.obs: ctm.chunk.u(cmbl, keypress, 49, 1)
ctm.obs: ctm.chunk.j(cmbl, keypress, 49, 1)
ctm.obs: ctm.chunk.c.l([[cmbl, keypress, 49, 1]])
ctm.obs: ctm.chunk.u(femx, keypress, 50, 0.800000)
ctm.obs: ctm.chunk.j(femx, keypress, 50, 0.800000)
ctm.obs: ctm.chunk.c.l([[femx, keypress, 50, 0.800000], [cmbl, keypress, 49, 0.900000]])
ctm.obs: ctm.chunk.u(dmhv, keypress, 51, 0.700000)
ctm.obs: ctm.chunk.j(dmhv, keypress, 51, 0.700000)
ctm.obs: ctm.chunk.u(npyi, keypress, 52, 0.700000)
ctm.obs: ctm.chunk.c.l([[femx, keypress, 50, 0.700000], [dmhv, keypress, 51, 0.700000], [cmbl, keypress, 49, 0.800000]])
ctm.obs: ctm.chunk.u(ejio, keypress, 53, 0.600000)
ctm.obs: ctm.chunk.j(ejio, keypress, 53, 0.600000)
ctm.obs: ctm.chunk.j(npyi, keypress, 52, 0.700000)
ctm.obs: ctm.chunk.c.l([[femx, keypress, 50, 0.600000], [dmhv, keypress, 51, 0.600000], [ejio, keypress, 53, 0.600000], [cmbl,
      keypress, 49, 0.700000], [npyi, keypress, 52, 0.700000]])
ctm.obs: ctm.chunk.u(nayx, keypress, 54, 0.700000)
ctm.obs: ctm.chunk.j(nayx, keypress, 54, 0.700000)
ctm.obs: ctm.chunk.c.l([[femx, keypress, 50, 0.500000], [dmhv, keypress, 51, 0.500000], [ejio, keypress, 53, 0.500000], [cmbl,
      keypress, 49, 0.600000], [npyi, keypress, 52, 0.600000], [nayx, keypress, 54, 0.700000]])
ctm.obs: ctm.chunk.u(glkl, keypress, 55, 0.900000)
ctm.obs: ctm.chunk.j(glkl, keypress, 55, 0.900000)
ctm.obs: ctm.chunk.c.l([[femx, keypress, 50, 0.400000], [dmhv, keypress, 51, 0.400000], [ejio, keypress, 53, 0.400000], [cmbl,
      keypress, 49, 0.500000], [npyi, keypress, 52, 0.500000], [nayx, keypress, 54, 0.600000], [glkl, keypress, 55,
     0.900000]])
ctm.obs: ctm.chunk.u(ulai, keypress, 56, 1)
ctm.obs: ctm.chunk.u(kera, keypress, 57, 0.900000)
ctm.obs: ctm.chunk.d(femx, keypress, 50, 0.300000)
ctm.obs: ctm.chunk.d(dmhv, keypress, 51, 0.300000)
ctm.obs: ctm.chunk.j(kera, keypress, 57, 0.900000)
ctm.obs: ctm.chunk.j(ulai, keypress, 56, 1)
ctm.obs: ctm.chunk.c.l([[ejio, keypress, 53, 0.300000], [cmbl, keypress, 49, 0.400000], [npyi, keypress, 52, 0.400000], [nayx,
      keypress, 54, 0.500000], [glkl, keypress, 55, 0.800000], [kera, keypress, 57, 0.900000], [ulai, keypress, 56, 1]])
ctm.obs: ctm.chunk.u(onbj, keypress, 48, 1)
ctm.obs: ctm.chunk.d(ejio, keypress, 53, 0.200000)
ctm.obs: ctm.chunk.j(onbj, keypress, 48, 1)
ctm.obs: ctm.chunk.c.l([[cmbl, keypress, 49, 0.300000], [npyi, keypress, 52, 0.300000], [nayx, keypress, 54, 0.400000], [glkl,
      keypress, 55, 0.700000], [kera, keypress, 57, 0.800000], [ulai, keypress, 56, 0.900000], [onbj, keypress, 48, 1]])
ctm.obs: ctm.chunk.c.l([[cmbl, keypress, 49, 0.200000], [npyi, keypress, 52, 0.200000], [nayx, keypress, 54, 0.300000], [glkl,
      keypress, 55, 0.600000], [kera, keypress, 57, 0.700000], [ulai, keypress, 56, 0.800000], [onbj, keypress, 48,
     0.900000]])
ctm.obs: ctm.chunk.c.l([[cmbl, keypress, 49, 0.100000], [npyi, keypress, 52, 0.100000], [nayx, keypress, 54, 0.200000], [glkl,
      keypress, 55, 0.500000], [kera, keypress, 57, 0.600000], [ulai, keypress, 56, 0.700000], [onbj, keypress, 48,
     0.800000]])
ctm.obs: ctm.chunk.c.l([[cmbl, keypress, 49, 0], [npyi, keypress, 52, 0], [nayx, keypress, 54, 0.100000], [glkl, keypress, 55,
      0.400000], [kera, keypress, 57, 0.500000], [ulai, keypress, 56, 0.600000], [onbj, keypress, 48, 0.700000]])
ctm.obs: ctm.chunk.d(cmbl, keypress, 49, -0.100000)
ctm.obs: ctm.chunk.d(npyi, keypress, 52, -0.100000)
ctm.obs: ctm.chunk.c.l([[nayx, keypress, 54, 0], [glkl, keypress, 55, 0.300000], [kera, keypress, 57, 0.400000], [ulai,
     keypress, 56, 0.500000], [onbj, keypress, 48, 0.600000]])
ctm.obs: ctm.chunk.d(nayx, keypress, 54, -0.100000)
ctm.obs: ctm.chunk.c.l([[glkl, keypress, 55, 0.200000], [kera, keypress, 57, 0.300000], [ulai, keypress, 56, 0.400000], [onbj,
      keypress, 48, 0.500000]])
ctm.obs: ctm.chunk.c.l([[glkl, keypress, 55, 0.100000], [kera, keypress, 57, 0.200000], [ulai, keypress, 56, 0.300000], [onbj,
      keypress, 48, 0.400000]])
```

```
ctm.obs: ctm.chunk.c.l([[glkl, keypress, 55, 0], [kera, keypress, 57, 0.100000], [ulai, keypress, 56, 0.200000], [onbj,
     keypress, 48, 0.300000]])
ctm.obs: ctm.chunk.d(glkl, keypress, 55, -0.100000)
ctm.obs: ctm.chunk.c.l([[kera, keypress, 57, 0], [ulai, keypress, 56, 0.100000], [onbj, keypress, 48, 0.200000]])
ctm.obs: ctm.chunk.d(kera, keypress, 57, -0.100000)
ctm.obs: ctm.chunk.c.l([[ulai, keypress, 56, 0], [onbj, keypress, 48, 0.100000]])
ctm.obs: ctm.chunk.d(ulai, keypress, 56, -0.100000)
ctm.obs: ctm.chunk.c.l([[onbj, keypress, 48, 0]])
ctm.obs: ctm.chunk.d(onbj, keypress, 48, -0.100000)
```

When the following two *chunks* are emitted by the actor:

```
ctm.obs: ctm.chunk.u(ulai, keypress, 56, 1)
ctm.obs: ctm.chunk.u(kera, keypress, 57, 0.900000)
```

while the *stage* is at capacity, we can see that the two *chunks* on *stage* with the lowest weight get dropped so that the two new *chunks* can get into the stage:

```
ctm.obs: ctm.chunk.d(femx, keypress, 50, 0.300000)
ctm.obs: ctm.chunk.d(dmhv, keypress, 51, 0.300000)
ctm.obs: ctm.chunk.j(kera, keypress, 57, 0.900000)
ctm.obs: ctm.chunk.j(ulai, keypress, 56, 1)
ctm.obs: ctm.chunk.c.l([[ejio, keypress, 53, 0.300000], [cmbl, keypress, 49, 0.400000], [npyi, keypress, 52, 0.400000], [nayx,
      keypress, 54, 0.500000], [glkl, keypress, 55, 0.800000], [kera, keypress, 57, 0.900000], [ulai, keypress, 56, 1]])
```

Memorizing numbers by *conscious* rehearsal

As a first example of a CTM, we are going to look at simulating something that we all do (or did at some point) frequently: memorizing one (or more) numbers by rehearsing. This will involve: having a way to bring a number into *consciouness* and trying to keep it there long enough for it to be retained permanently by an *actor* setup to recall numbers. Because this is an exploration of CTM, and not a realistic simulation, we will not requires the rehearshed numbers to be rehearshed for a human realistic time. We will, however, build a way to simulate *distractions* so that numbers may get dropped from the *stage* and thus possibly impact rehearsing in a way that may lead to numbers getting forgotten.

To get started, let's look first at how we would build *distraction* in such a scenario. All we need, is for an *actor* to emit a *chunk* with a weight high enough to get on *stage*. The more of such *chunks* on *stage*, the more load on the *consciouness* there will be and the more likely *chunks* holding the numbers we are trying to rehearsh will get drop off (depending on the weights).

Let's create a new *fizz* file called `ruffle.fizz`. We will put in it the definition for our *distraction actor* which we will call `ctm.actor.ruffle`:

```
ctm.actor.ruffle {

    ()  :-  @console.keypress(114),
            rnd.real(1,:w,$ruffle.weight.min,$ruffle.weight.max),
            declare(ctm.chunk.u(%sym.4,ruffle,114,:w)),
            hush;

}
```

Compared to what we have seen when creating the *stage* implementation, the *procedural knowledge* for the *actor* is fairly simple. When the user will press the `R` key on the keyboard, we will pick a random weight value for the *chunk* within a range that is specified by the *constants* `$ruffle.weight.min` and `$ruffle.weight.max`. Then we will then use the *primitive* `declare` to send the *chunk* to the *stage* and since we are assiging a random GUID (with the *volatile* `sym.4`) for each *chunks*, multiple keypress events will generate as many *chunks* as desired.

Before we can test this, we need to create a new *solution file*. Let's call it `memorize.json`. Copy and paste the following into it:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["stage.fizz", "debug.fizz", "ticks.fizz", "ruffle.fizz" ],
        "globals" :   [
            {
                "label" : "tick.hush",
                "value" : 5
            },
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            },
            {
                "label" : "stage.size",
                "value" : 7
            },
            {
                "label" : "stage.loss",
                "value" : 0.1
            },
            {
                "label" : "stage.drop",
                "value" : -0.1
            },
            {
                "label" : "ruffle.weight.min",
                "value" : 0.7
            },
            {
                "label" : "ruffle.weight.max",
                "value" : 1.0
            }
        ]
    }
}
```

At the bottom of it, we have added the values for the ruffle's weight range. Any of the *chunks* we will be pushing onto the *stage* will have a weight no smaller than `0.7`. This will insure that our *actor* do cause some *distraction*. We can then load the solution and press the `R` key a couple of time to check that it is working:

```
$ ./fizz.x64 ./etc/experiments/ctm/memorize.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/memorize.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.009s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.001s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.001s
load : loading ./etc/experiments/ctm/number.fizz ...
load : loaded ./etc/experiments/ctm/number.fizz in 0.003s
load : loading ./etc/experiments/ctm/memorize.fizz ...
load : loaded ./etc/experiments/ctm/memorize.fizz in 0.003s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.028s
load : loading completed in 0.031s
ctm.obs: ctm.chunk.u(erhs, ruffle, 114, 0.840543)
ctm.obs: ctm.chunk.j(erhs, ruffle, 114, 0.840543)
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.840543]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.740543]])
ctm.obs: ctm.chunk.u(axxl, ruffle, 114, 0.973471)
ctm.obs: ctm.chunk.j(axxl, ruffle, 114, 0.973471)
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.640543], [axxl, ruffle, 114, 0.973471]])
ctm.obs: ctm.chunk.u(gwva, ruffle, 114, 0.882959)
ctm.obs: ctm.chunk.j(gwva, ruffle, 114, 0.882959)
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.540543], [axxl, ruffle, 114, 0.873471], [gwva, ruffle, 114, 0.882959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.440543], [axxl, ruffle, 114, 0.773471], [gwva, ruffle, 114, 0.782959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.340543], [axxl, ruffle, 114, 0.673471], [gwva, ruffle, 114, 0.682959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.240543], [axxl, ruffle, 114, 0.573471], [gwva, ruffle, 114, 0.582959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.140543], [axxl, ruffle, 114, 0.473471], [gwva, ruffle, 114, 0.482959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, 0.040543], [axxl, ruffle, 114, 0.373471], [gwva, ruffle, 114, 0.382959]])
ctm.obs: ctm.chunk.c.l([[erhs, ruffle, 114, -0.059457], [axxl, ruffle, 114, 0.273471], [gwva, ruffle, 114, 0.282959]])
ctm.obs: ctm.chunk.d(erhs, ruffle, 114, -0.159457)
ctm.obs: ctm.chunk.c.l([[axxl, ruffle, 114, 0.173471], [gwva, ruffle, 114, 0.182959]])
ctm.obs: ctm.chunk.c.l([[axxl, ruffle, 114, 0.073471], [gwva, ruffle, 114, 0.082959]])
ctm.obs: ctm.chunk.c.l([[axxl, ruffle, 114, -0.026529], [gwva, ruffle, 114, -0.017041]])
ctm.obs: ctm.chunk.d(axxl, ruffle, 114, -0.126529)
ctm.obs: ctm.chunk.d(gwva, ruffle, 114, -0.117041)
```

As we said earlier, we need to have an *actor* that when requested will (attempt) to push onto the *stage* a number. For this, we will adapt the keypress *actor* we have used previously. Create a new *fizz* file, `number.fizz` then copy and paste the following code into it:

```
ctm.actor.number.input {

    weight.value = 0.5,
    weight.range = <0.2|1>,
    weight.loss  = 0.1,
    started      = no

} {

    // send a number for every correct keypress we get
    ()  :-  @console.keypress(:k?[<48|57>]),
            sub(:k,48,:n),
            declare(ctm.chunk.u(%sym.4,number,:n,$weight.value)),
            poke(started,yes),
            hush;

    // lower the weight as time flow
    ()  :-  @ctm.tick.fast(_,_),
            peek(started,yes),
            sub($weight,$weight.loss,:w.o),
            rng.clamp($weight.range,:w.o,:w.c),
            poke(weight.value,:w.c),
            hush;

    // each time one of our chunk get "on stage" we increase the weight we will use for the next chunk we try pushing
    ()  :-  @ctm.chunk.j(_,number,_,_),
            add($weight,$loss,:w.o),
            rng.clamp($weight.range,:w.o,:w.c),
            poke(weight.value,:w.c),
            hush;

}
```

Just like for the keypress *actor*, this *actor* will emit a *chunk* at each keypress while the weight value it will be using will go down as time flows and go up whenever one of its *chunks* get onto the *stage*. However, instead of accepting any keys, a *constraint* is set on the value that would unify with the *variable* `k`. This will insure that only the keys `0` to `9` are able to trigger inferring over the *prototype*. We then substract (with the *primitive* `sub`) `48` from the keycode so that each of the *chunks* contains the single digit number we are trying to push onto the *stage*.

Once we add the file `number.fizz` to the list of files to be loaded in `memorize.json`, we can try it out by pressing the key `5`:

```
$ ./fizz.x64 ./etc/experiments/ctm/memorize.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt
```

```
load : loading ./etc/experiments/ctm/memorize.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.009s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.001s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.001s
load : loading ./etc/experiments/ctm/number.fizz ...
load : loaded ./etc/experiments/ctm/number.fizz in 0.003s
load : loading ./etc/experiments/ctm/memorize.fizz ...
load : loaded ./etc/experiments/ctm/memorize.fizz in 0.003s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.028s
load : loading completed in 0.031s
ctm.obs: ctm.chunk.u(lvvt, number, 5, 0.500000)
ctm.obs: ctm.chunk.j(lvvt, number, 5, 0.500000)
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0.500000]])
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0.400000]])
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0.300000]])
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0.200000]])
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0.100000]])
ctm.obs: ctm.chunk.c.l([[lvvt, number, 5, 0]])
ctm.obs: ctm.chunk.d(lvvt, number, 5, -0.100000)
```

Just as we observed before, the *chunk* get on *stage* then gets removed once its weight dropped to the *stage* threshold.

To continue, create a new *fizz* file called `memorize.fizz`. In it, we are going to define the *actor* that will memorize the numbers. You will also need to add that file to `memorize.json` so that it get loaded as well.

The class of the *elemental* we are going to use to memorize numbers is `MRKCLettered`. Such class is built to only manage *statements* which will be the case here, but it also support an optional short-term/long-term storage which fits our needs here. Let's look at the definition of the *elemental* as we are going to use it:

```
ctm.actor.number {

    class      = MRKCLettered,
    recall.frq = $tick.hush,
    recall.ttl = $tick.slow,
    recall.add = $tick.slow,
    recall.thd = $tick.dull

} {}
```

As you can see, we are using four built-in properties of that class of *elementals* to setup the *recall* feature (using the tick constants we have already defined in our solution files). This class works by setting an *time-to-live* value (using `recall.ttl`) to every *statements* that get asserted with a valid timestamp property. Then each time the *statement* is used as part of a response to a query that the *elemental* gets, the *time-to-live* value gets increased (by the value set in `recall.add`). If the value gets above a threshold value (`recall.thd`), the *statement* get commited to permanent storage. Every so often (as set by `recall.frq`), the *elemental* looks at all the *statements* it has and remove all the ones whose time have expired. In this example, we have used for settings the various tick values. This will ensure that under normal conditions a number will get stored in long-term memory in only 3 to 4 queries (about two seconds).

To interface this with the *stage* we need to create an *actor* which will assert (which unlike `declare` insures that the *statement* will be retained by an *elemental*, creating one if needed) a `ctm.actor.number` *statement* when a new number shows-up on *stage*, then query it each times the corresponding *chunk* is on *stage*:

```
ctm.actor.number.rehears {

    // when a number first show-up on stage, we assert it into the store (with a time stamp)
    ()  :-  @ctm.chunk.j(_,number,:d,_),
            assert(ctm.actor.number(:d),1.0,{stp = %now}),
            hush;

```

```
 8     // when a number is on stage, we query the store (in order to reharse the number)
 9     ()  :-  @ctm.chunk.c(_,number,:d,_),
10             #ctm.actor.number(:d),
11             hush;
12
13 }
```

Because, the class `MRKCLettered` only enable its recall feature on a *statement* if it has a timestamp as part of its properties, we specify a *frame* as the last *term* of the *primitive* `assert` to set the timestamp (using the volatile `now`) on line 5.

We are now ready to try this out, starting with a simple case where we try to recall one number. Note that we will be using the console *command* `spy` to observe what happens to the *statement* storing the number as we will be trying to reharse it. Make sure you are typing that *command* before pressing one of the number key:

```
$ ./fizz.x64 ./etc/experiments/ctm/memorize.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/memorize.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.007s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.002s
load : loading ./etc/experiments/ctm/number.fizz ...
load : loaded ./etc/experiments/ctm/number.fizz in 0.004s
load : loading ./etc/experiments/ctm/memorize.fizz ...
load : loaded ./etc/experiments/ctm/memorize.fizz in 0.005s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.039s
load : loading completed in 0.042s
?- /spy(append,ctm.actor.number)
spy : observing ctm.actor.number
ctm.obs: ctm.chunk.u(pmxj, number, 7, 0.500000)
ctm.obs: ctm.chunk.j(pmxj, number, 7, 0.500000)
spy : S ctm.actor.number(7) {stp = 1542214225.950980, ttl = 1} (90.000000)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0.500000]])
spy : Q #ctm.actor.number(7) (89.999809)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 2} (89.999763)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0.400000]])
spy : Q #ctm.actor.number(7) (89.999809)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 3} (89.999733)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0.300000]])
spy : Q #ctm.actor.number(7) (89.999825)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 4} (89.999763)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0.200000]])
spy : Q #ctm.actor.number(7) (89.999794)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 5} (89.999733)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0.100000]])
spy : Q #ctm.actor.number(7) (89.999817)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 6} (89.999733)
ctm.obs: ctm.chunk.c.l([[pmxj, number, 7, 0]])
ctm.obs: ctm.chunk.d(pmxj, number, 7, -0.100000)
```

If we query the *elemental* at this point, we will get the *statement* containing the number which have been memorized:

```
?- #ctm.actor.number(:x)
spy : Q #ctm.actor.number(:x) (89.999886)
spy : R ctm.actor.number(7) {stp = 1542214225.950980, ttl = 0} (89.999741)
-> ( 7 ) := 1.00 (0.001) 1
```

Note that the `ttl` property is `0`. This indicate that the number is now permanently stored. Let see now what happens when we are trying the memorize a number while there's a large amount of distractions. We will use the *ruffle actor* that we added earlier (bound to the key `R`):

```
ctm.obs: ctm.chunk.u(kvbb, ruffle, 114, 0.949761)
ctm.obs: ctm.chunk.u(lrih, ruffle, 114, 0.899560)
ctm.obs: ctm.chunk.u(sepq, ruffle, 114, 0.958070)
ctm.obs: ctm.chunk.j(lrih, ruffle, 114, 0.899560)
ctm.obs: ctm.chunk.j(kvbb, ruffle, 114, 0.949761)
ctm.obs: ctm.chunk.j(sepq, ruffle, 114, 0.958070)
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.899560], [kvbb, ruffle, 114, 0.949761], [sepq, ruffle, 114, 0.958070]])
ctm.obs: ctm.chunk.u(ljuw, number, 5, 0.500000)
ctm.obs: ctm.chunk.j(ljuw, number, 5, 0.500000)
ctm.obs: ctm.chunk.c.l([[ljuw, number, 5, 0.500000], [lrih, ruffle, 114, 0.799560], [kvbb, ruffle, 114, 0.849761], [sepq,
      ruffle, 114, 0.858070]])
ctm.obs: ctm.chunk.u(hkcw, ruffle, 114, 0.811822)
ctm.obs: ctm.chunk.u(benv, ruffle, 114, 0.849987)
ctm.obs: ctm.chunk.j(hkcw, ruffle, 114, 0.811822)
ctm.obs: ctm.chunk.j(benv, ruffle, 114, 0.849987)
ctm.obs: ctm.chunk.u(cvoo, ruffle, 114, 0.872611)
ctm.obs: ctm.chunk.c.l([[ljuw, number, 5, 0.400000], [lrih, ruffle, 114, 0.699560], [kvbb, ruffle, 114, 0.749761], [sepq,
      ruffle, 114, 0.758070], [hkcw, ruffle, 114, 0.811822], [benv, ruffle, 114, 0.849987]])
ctm.obs: ctm.chunk.u(osao, ruffle, 114, 0.908849)
ctm.obs: ctm.chunk.d(ljuw, number, 5, 0.300000)
ctm.obs: ctm.chunk.j(cvoo, ruffle, 114, 0.872611)
ctm.obs: ctm.chunk.j(osao, ruffle, 114, 0.908849)
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.599560], [kvbb, ruffle, 114, 0.649761], [sepq, ruffle, 114, 0.658070], [hkcw,
      ruffle, 114, 0.711822], [benv, ruffle, 114, 0.749987], [cvoo, ruffle, 114, 0.872611], [osao, ruffle, 114, 0.908849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.499560], [kvbb, ruffle, 114, 0.549761], [sepq, ruffle, 114, 0.558070], [hkcw,
      ruffle, 114, 0.611822], [benv, ruffle, 114, 0.649987], [cvoo, ruffle, 114, 0.772611], [osao, ruffle, 114, 0.808849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.399560], [kvbb, ruffle, 114, 0.449761], [sepq, ruffle, 114, 0.458070], [hkcw,
      ruffle, 114, 0.511822], [benv, ruffle, 114, 0.549987], [cvoo, ruffle, 114, 0.672611], [osao, ruffle, 114, 0.708849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.299560], [kvbb, ruffle, 114, 0.349761], [sepq, ruffle, 114, 0.358070], [hkcw,
      ruffle, 114, 0.411822], [benv, ruffle, 114, 0.449987], [cvoo, ruffle, 114, 0.572611], [osao, ruffle, 114, 0.608849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.199560], [kvbb, ruffle, 114, 0.249761], [sepq, ruffle, 114, 0.258070], [hkcw,
      ruffle, 114, 0.311822], [benv, ruffle, 114, 0.349987], [cvoo, ruffle, 114, 0.472611], [osao, ruffle, 114, 0.508849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, 0.099560], [kvbb, ruffle, 114, 0.149761], [sepq, ruffle, 114, 0.158070], [hkcw,
      ruffle, 114, 0.211822], [benv, ruffle, 114, 0.249987], [cvoo, ruffle, 114, 0.372611], [osao, ruffle, 114, 0.408849]])
ctm.obs: ctm.chunk.c.l([[lrih, ruffle, 114, -0.000440], [kvbb, ruffle, 114, 0.049761], [sepq, ruffle, 114, 0.058070], [hkcw,
      ruffle, 114, 0.111822], [benv, ruffle, 114, 0.149987], [cvoo, ruffle, 114, 0.272611], [osao, ruffle, 114, 0.308849]])
ctm.obs: ctm.chunk.d(lrih, ruffle, 114, -0.100440)
ctm.obs: ctm.chunk.c.l([[kvbb, ruffle, 114, -0.050239], [sepq, ruffle, 114, -0.041930], [hkcw, ruffle, 114, 0.011822], [benv,
      ruffle, 114, 0.049987], [cvoo, ruffle, 114, 0.172611], [osao, ruffle, 114, 0.208849]])
ctm.obs: ctm.chunk.d(kvbb, ruffle, 114, -0.150239)
ctm.obs: ctm.chunk.d(sepq, ruffle, 114, -0.141930)
ctm.obs: ctm.chunk.c.l([[hkcw, ruffle, 114, -0.088178], [benv, ruffle, 114, -0.050013], [cvoo, ruffle, 114, 0.072611], [osao,
      ruffle, 114, 0.108849]])
ctm.obs: ctm.chunk.d(hkcw, ruffle, 114, -0.188178)
ctm.obs: ctm.chunk.d(benv, ruffle, 114, -0.150013)
ctm.obs: ctm.chunk.c.l([[cvoo, ruffle, 114, -0.027389], [osao, ruffle, 114, 0.008849]])
ctm.obs: ctm.chunk.d(cvoo, ruffle, 114, -0.127389)
ctm.obs: ctm.chunk.c.l([[osao, ruffle, 114, -0.091151]])
ctm.obs: ctm.chunk.d(osao, ruffle, 114, -0.191151)
?- #ctm.actor.number(:x)
```

In this instance, we have loaded the *stage* with three *distraction chunks* before emitting the *chunk* that hold the number. We then add some more *ruffles* which cause the number *chunk* to get removed. Later on, we query the `ctm.actor.number` to see if the number was memorized and we do not get any answers, as expected.

Memorizing a list of numbers by *conscious* rehearsal

Let's build upon the previous example and extend it to memorize numbers as a collection. This mean that each time a number appears *on stage* we will add it a *list* that is being reharsed as we did for the numbers.

Let start by creating a new *fizz* file `memlist.fizz` as well as new solution file `memlist.json`. In the JSON file we will be reusing the JSON from the previous example, only replacing the name of the last *fizz* file to be loaded:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["stage.fizz", "debug.fizz", "ticks.fizz", "ruffle.fizz", "number.fizz", "memlist.fizz" ],
        "globals" :   [
            {
                "label" : "tick.hush",
                "value" : 5
            },
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            },
            {
                "label" : "stage.size",
                "value" : 7
            },
            {
                "label" : "stage.loss",
                "value" : 0.1
            },
            {
                "label" : "stage.drop",
                "value" : -0.1
            },
            {
                "label" : "ruffle.weight.min",
                "value" : 0.7
            },
            {
                "label" : "ruffle.weight.max",
                "value" : 1.0
            }
        ]
    }
}
```

In `memlist.fizz`, we are going first to define the *elemental* that will hold the *list(s)*, just like we did for memorizing numbers:

```
ctm.actor.list.store {

    class      = MRKCLettered,
    recall.frq = $tick.hush,
    recall.ttl = $tick.slow,
    recall.add = $tick.slow,
    recall.thd = $tick.dull

} {}
```

We will then add an new *actor* (call it `ctm.actor.list.logic`) which will be the interface between the store, the *stage* and the user inputs. Here is its definition:

```
ctm.actor.list.logic {

    key     = nil,
    weight  = 0.6

} {

    // when a new number shows up on stage, and we have no on-going list we create it
    ()  :-  @ctm.chunk.j(_,number,:d,_),
            peek(key,nil)^,
            set(:k,%sym.4),
            poke(key,:k),
            assert(ctm.actor.list.store(:k,[:d]),1.0,{stp = %now, ttl = 1}),
            declare(ctm.chunk.u(:k,list,[:d],$weight)),
            hush;

    // when a new number shows up on stage, and we have we add it to the list
18      ()  :-  @ctm.chunk.j(_,number,:d,_),
19              peek(key,:k?[neq(nil)])^,
20              #ctm.actor.list.store(:k,:l),
21              change([ctm.actor.list.store(:k,:l)],[ctm.actor.list.store(:k,[:d|:l]),1.0,{stp = %now}]),
22              declare(ctm.chunk.u(:k,list,[:d|:l],$weight)),
23              hush;
24
25      // for each one of our chunk broadcasted by the stage, we rehears the list so that it don't get forgotten
26      ()  :-  @ctm.chunk.c($key,list,_,_),
27              #ctm.actor.list.store($key,_),
28              hush;
29
30      // when the chunk that hold the list, is removed from the stage, we reset the property
31      ()  :-  @ctm.chunk.d($key,list,_,_),
32              poke(key,nil),
33              hush;
34
35  }
```

It contains four *prototypes* dealing with three events: a new number shows-up on *stage*, our *list* is on *stage* and dropped from the *stage*. The same *trigger predicate* is used on the first two *predicates*. This is necessary to handle the creation of a new *list* when we are not already managing one. We use the property `key` (which is initialized as `nil`) to keep a GUID associated with the *list* being filled. We use the very same GUID as the first *term* of the `ctm.actor.list.store` *statement* we will be asserting. Note that contrary to our previous example, we are this time providing a *time-to-live* (the `ttl` property in the *statement*) value along with the timestamp. The first *prototype* completes with emitting a *chunk* for the *list*. The second *prototype* get executed also when a number first shows-up on *stage* but only carry on if a *list* has been started (by checking that the `key` property isn't the `nil` value). It then query `ctm.actor.list.store` for the *list*, and uses the *primitive* `change` to replace the *statement* holding the *list* by a new *statement* that include the new number. Finally, the *chunk* associated with the *list* get emitted so that its weight get reset since it is slowly decaying as the *chunk* sits on *stage*. On Line 26, the third *prototype* get triggered when the *chunk* for the *list* is broadcasted as being *on stage*, and as we have done before it query `ctm.actor.list.store` so that the *time-to-live* value of the *statement* increase. The last *prototype* resets the `key` property when the *chunk* is removed from the *stage*.

Let's load-up the solution now and see how that works when we try to memorize a list of three numbers. Note that here again, we will use the console *command* `spy` to observe `ctm.actor.list.store`:

```
$ ./fizz.x64 ./etc/experiments/ctm/memlist.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/memlist.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.006s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.002s
load : loading ./etc/experiments/ctm/number.fizz ...
load : loaded ./etc/experiments/ctm/number.fizz in 0.003s
load : loading ./etc/experiments/ctm/memlist.fizz ...
load : loaded ./etc/experiments/ctm/memlist.fizz in 0.007s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.030s
load : loading completed in 0.032s
?- /spy(append,ctm.actor.list.store)
spy : observing ctm.actor.list.store
ctm.obs: ctm.chunk.u(jvvp, number, 1, 0.500000)
ctm.obs: ctm.chunk.j(jvvp, number, 1, 0.500000)
ctm.obs: ctm.chunk.u(syva, list, [1], 0.600000)
spy : Q #ctm.actor.list.store(syva, :l) (89.999062)
spy : S ctm.actor.list.store(syva, [1]) {stp = 1542223376.259085, ttl = 1} (90.000000)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0.500000]])
```

After the `spy` command, we pressed the `1` key which caused the number to be pushed onto the *stage*. The `ctm.chunk.j` *statement* that is broadcasted by the *stage* triggers our list *chunk* as well as as a *statement* `ctm.actor.list.store`. However, the new *chunk* hasn't been taken yet in consideration by the *stage* as we can see in the `ctm.chunk.c.l` *statement*.

```
ctm.obs: ctm.chunk.u(dyct, number, 2, 0.500000)
ctm.obs: ctm.chunk.j(dyct, number, 2, 0.500000)
spy : Q #ctm.actor.list.store(syva, :l) (89.999535)
spy : R ctm.actor.list.store(syva, [1]) {stp = 1542223376.259085, ttl = 2} (89.999496)
ctm.obs: ctm.chunk.j(syva, list, [1], 0.600000)
ctm.obs: ctm.chunk.u(syva, list, [2, 1], 0.600000)
spy : S ctm.actor.list.store(syva, [2, 1]) {stp = 1542223377.262067, ttl = 1} (90.000000)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0.400000], [dyct, number, 2, 0.500000], [syva, list, [1], 0.600000]])
```

As we add a new number to the *stage*, we observe our *list* being updated to include it, even though the *stage* is getting broadcasted with an older version of the *chunk*.

```
ctm.obs: ctm.chunk.u(jdle, number, 3, 0.500000)
ctm.obs: ctm.chunk.j(jdle, number, 3, 0.500000)
spy : Q #ctm.actor.list.store(syva, :l) (89.999527)
spy : R ctm.actor.list.store(syva, [2, 1]) {stp = 1542223377.262067, ttl = 2} (89.999489)
ctm.obs: ctm.chunk.u(syva, list, [3, 2, 1], 0.600000)
spy : Q #ctm.actor.list.store(syva, _) (89.999634)
spy : S ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 2} (90.000000)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 2} (89.999519)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0.300000], [dyct, number, 2, 0.400000], [jdle, number, 3, 0.500000], [syva, list,
      [2, 1], 0.600000]])
spy : Q #ctm.actor.list.store(syva, _) (89.999794)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 3} (89.999741)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0.200000], [dyct, number, 2, 0.300000], [jdle, number, 3, 0.400000], [syva, list,
      [3, 2, 1], 0.600000]])
spy : Q #ctm.actor.list.store(syva, _) (89.999779)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 4} (89.999733)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0.100000], [dyct, number, 2, 0.200000], [jdle, number, 3, 0.300000], [syva, list,
      [3, 2, 1], 0.500000]])
```

Eventually though, the *stage*'s contents reflect the *list* as we can see above.

```
spy : Q #ctm.actor.list.store(syva, _) (89.999771)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 5} (89.999718)
ctm.obs: ctm.chunk.c.l([[jvvp, number, 1, 0], [dyct, number, 2, 0.100000], [jdle, number, 3, 0.200000], [syva, list, [3, 2,
      1], 0.400000]])
ctm.obs: ctm.chunk.d(jvvp, number, 1, -0.100000)
spy : Q #ctm.actor.list.store(syva, _) (89.999809)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 6} (89.999771)
ctm.obs: ctm.chunk.c.l([[dyct, number, 2, 0], [jdle, number, 3, 0.100000], [syva, list, [3, 2, 1], 0.300000]])
ctm.obs: ctm.chunk.d(dyct, number, 2, -0.100000)
spy : Q #ctm.actor.list.store(syva, _) (89.999794)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 0} (89.999741)
ctm.obs: ctm.chunk.c.l([[jdle, number, 3, 0], [syva, list, [3, 2, 1], 0.200000]])
ctm.obs: ctm.chunk.d(jdle, number, 3, -0.100000)
spy : Q #ctm.actor.list.store(syva, _) (89.999756)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 0} (89.999702)
ctm.obs: ctm.chunk.c.l([[syva, list, [3, 2, 1], 0.100000]])
spy : Q #ctm.actor.list.store(syva, _) (89.999733)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 0} (89.999680)
ctm.obs: ctm.chunk.c.l([[syva, list, [3, 2, 1], 0]])
ctm.obs: ctm.chunk.d(syva, list, [3, 2, 1], -0.100000)
?- #ctm.actor.list.store(:k,:l)
spy : Q #ctm.actor.list.store(:k, :l) (89.999886)
spy : R ctm.actor.list.store(syva, [3, 2, 1]) {stp = 1542223378.261700, ttl = 0} (89.999710)
-> ( syva , [3, 2, 1] ) := 1.00 (0.001) 1
```

Here we waited for the *stage* to be empty again before querying the *list* store directly, but we could have done it much sooner.

Let see now what happens if we add *distractions* after each new numbers (1 to 7) we try to add to a *list*:

```
ctm.obs: ctm.chunk.u(ootu, number, 1, 0.500000)
ctm.obs: ctm.chunk.u(djod, ruffle, 114, 0.945456)
ctm.obs: ctm.chunk.u(akah, ruffle, 114, 0.858300)
ctm.obs: ctm.chunk.j(ootu, number, 1, 0.500000)
ctm.obs: ctm.chunk.j(akah, ruffle, 114, 0.858300)
ctm.obs: ctm.chunk.u(xosy, list, [1], 0.600000)
ctm.obs: ctm.chunk.j(djod, ruffle, 114, 0.945456)
ctm.obs: ctm.chunk.c.l([[ootu, number, 1, 0.500000], [akah, ruffle, 114, 0.858300], [djod, ruffle, 114, 0.945456]])
ctm.obs: ctm.chunk.u(ftwf, number, 2, 0.500000)
ctm.obs: ctm.chunk.u(fnfs, ruffle, 114, 0.716353)
ctm.obs: ctm.chunk.j(ftwf, number, 2, 0.500000)
ctm.obs: ctm.chunk.j(xosy, list, [1], 0.600000)
ctm.obs: ctm.chunk.j(fnfs, ruffle, 114, 0.716353)
ctm.obs: ctm.chunk.u(xosy, list, [2, 1], 0.600000)
ctm.obs: ctm.chunk.c.l([[ootu, number, 1, 0.400000], [ftwf, number, 2, 0.500000], [xosy, list, [1], 0.600000], [fnfs, ruffle,
     114, 0.716353], [akah, ruffle, 114, 0.758300], [djod, ruffle, 114, 0.845456]])
ctm.obs: ctm.chunk.u(lxqp, ruffle, 114, 0.942872)
ctm.obs: ctm.chunk.u(cyew, number, 3, 0.500000)
ctm.obs: ctm.chunk.d(ootu, number, 1, 0.300000)
ctm.obs: ctm.chunk.j(cyew, number, 3, 0.500000)
ctm.obs: ctm.chunk.u(xosy, list, [3, 2, 1], 0.600000)
ctm.obs: ctm.chunk.j(lxqp, ruffle, 114, 0.942872)
ctm.obs: ctm.chunk.c.l([[ftwf, number, 2, 0.400000], [cyew, number, 3, 0.500000], [xosy, list, [2, 1], 0.600000], [fnfs,
     ruffle, 114, 0.616353], [akah, ruffle, 114, 0.658300], [djod, ruffle, 114, 0.745456], [lxqp, ruffle, 114, 0.942872]])
ctm.obs: ctm.chunk.u(fsay, ruffle, 114, 0.812914)
ctm.obs: ctm.chunk.u(jgfw, ruffle, 114, 0.997678)
ctm.obs: ctm.chunk.d(ftwf, number, 2, 0.300000)
ctm.obs: ctm.chunk.d(cyew, number, 3, 0.400000)
ctm.obs: ctm.chunk.j(fsay, ruffle, 114, 0.812914)
ctm.obs: ctm.chunk.j(jgfw, ruffle, 114, 0.997678)
ctm.obs: ctm.chunk.c.l([[fnfs, ruffle, 114, 0.516353], [akah, ruffle, 114, 0.558300], [xosy, list, [3, 2, 1], 0.600000], [djod
     , ruffle, 114, 0.645456], [fsay, ruffle, 114, 0.812914], [lxqp, ruffle, 114, 0.842872], [jgfw, ruffle, 114, 0.997678]])
ctm.obs: ctm.chunk.u(wypf, number, 4, 0.500000)
ctm.obs: ctm.chunk.u(jpne, ruffle, 114, 0.941351)
ctm.obs: ctm.chunk.u(lxvq, ruffle, 114, 0.918608)
ctm.obs: ctm.chunk.d(fnfs, ruffle, 114, 0.416353)
ctm.obs: ctm.chunk.d(akah, ruffle, 114, 0.458300)
ctm.obs: ctm.chunk.d(xosy, list, [3, 2, 1], 0.500000)
```

Oopsy ... too much new *distractions* caused the *chunk* related to the *list* to get dropped from the *stage*. As new numbers get onto the *stage*, a new *list* is started:

```
ctm.obs: ctm.chunk.j(wypf, number, 4, 0.500000)
ctm.obs: ctm.chunk.u(yubh, list, [4], 0.600000)
ctm.obs: ctm.chunk.j(lxvq, ruffle, 114, 0.918608)
ctm.obs: ctm.chunk.j(jpne, ruffle, 114, 0.941351)
ctm.obs: ctm.chunk.c.l([[wypf, number, 4, 0.500000], [djod, ruffle, 114, 0.545456], [fsay, ruffle, 114, 0.712914], [lxqp,
     ruffle, 114, 0.742872], [jgfw, ruffle, 114, 0.897678], [lxvq, ruffle, 114, 0.918608], [jpne, ruffle, 114, 0.941351]])
ctm.obs: ctm.chunk.u(cbhi, number, 5, 0.500000)
ctm.obs: ctm.chunk.d(wypf, number, 4, 0.400000)
ctm.obs: ctm.chunk.d(djod, ruffle, 114, 0.445456)
ctm.obs: ctm.chunk.j(cbhi, number, 5, 0.500000)
ctm.obs: ctm.chunk.j(yubh, list, [4], 0.600000)
ctm.obs: ctm.chunk.u(yubh, list, [5, 4], 0.600000)
ctm.obs: ctm.chunk.c.l([[cbhi, number, 5, 0.500000], [yubh, list, [4], 0.600000], [fsay, ruffle, 114, 0.612914], [lxqp, ruffle
     , 114, 0.642872], [jgfw, ruffle, 114, 0.797678], [lxvq, ruffle, 114, 0.818608], [jpne, ruffle, 114, 0.841351]])
ctm.obs: ctm.chunk.u(ybgx, ruffle, 114, 0.838483)
ctm.obs: ctm.chunk.u(hpet, ruffle, 114, 0.987456)
ctm.obs: ctm.chunk.d(cbhi, number, 5, 0.400000)
ctm.obs: ctm.chunk.d(fsay, ruffle, 114, 0.512914)
ctm.obs: ctm.chunk.j(ybgx, ruffle, 114, 0.838483)
ctm.obs: ctm.chunk.j(hpet, ruffle, 114, 0.987456)
ctm.obs: ctm.chunk.c.l([[lxqp, ruffle, 114, 0.542872], [yubh, list, [5, 4], 0.600000], [jgfw, ruffle, 114, 0.697678], [lxvq,
     ruffle, 114, 0.718608], [jpne, ruffle, 114, 0.741351], [ybgx, ruffle, 114, 0.838483], [hpet, ruffle, 114, 0.987456]])
ctm.obs: ctm.chunk.u(pehn, number, 6, 0.500000)
ctm.obs: ctm.chunk.u(bdyp, ruffle, 114, 0.822495)
ctm.obs: ctm.chunk.u(acni, ruffle, 114, 0.849296)
ctm.obs: ctm.chunk.d(lxqp, ruffle, 114, 0.442872)
ctm.obs: ctm.chunk.d(yubh, list, [5, 4], 0.500000)
```

But the new *list* also get kicked from the *stage*.

```
ctm.obs: ctm.chunk.j(bdyp, ruffle, 114, 0.822495)
ctm.obs: ctm.chunk.j(acni, ruffle, 114, 0.849296)
ctm.obs: ctm.chunk.c.l([[jgfw, ruffle, 114, 0.597678], [lxvq, ruffle, 114, 0.618608], [jpne, ruffle, 114, 0.641351], [ybgx,
     ruffle, 114, 0.738483], [bdyp, ruffle, 114, 0.822495], [acni, ruffle, 114, 0.849296], [hpet, ruffle, 114, 0.887456]])
ctm.obs: ctm.chunk.u(tuvv, number, 7, 0.500000)
ctm.obs: ctm.chunk.u(vgaa, ruffle, 114, 0.842811)
ctm.obs: ctm.chunk.d(jgfw, ruffle, 114, 0.497678)
ctm.obs: ctm.chunk.j(vgaa, ruffle, 114, 0.842811)
ctm.obs: ctm.chunk.c.l([[lxvq, ruffle, 114, 0.518608], [jpne, ruffle, 114, 0.541351], [ybgx, ruffle, 114, 0.638483], [bdyp,
     ruffle, 114, 0.722495], [acni, ruffle, 114, 0.749296], [hpet, ruffle, 114, 0.787456], [vgaa, ruffle, 114, 0.842811]])
ctm.obs: ctm.chunk.u(uuqr, ruffle, 114, 0.847225)
ctm.obs: ctm.chunk.d(lxvq, ruffle, 114, 0.418608)
ctm.obs: ctm.chunk.j(uuqr, ruffle, 114, 0.847225)
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, 0.441351], [ybgx, ruffle, 114, 0.538483], [bdyp, ruffle, 114, 0.622495], [acni,
     ruffle, 114, 0.649296], [hpet, ruffle, 114, 0.687456], [vgaa, ruffle, 114, 0.742811], [uuqr, ruffle, 114, 0.847225]])
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, 0.341351], [ybgx, ruffle, 114, 0.438483], [bdyp, ruffle, 114, 0.522495], [acni,
     ruffle, 114, 0.549296], [hpet, ruffle, 114, 0.587456], [vgaa, ruffle, 114, 0.642811], [uuqr, ruffle, 114, 0.747225]])
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, 0.241351], [ybgx, ruffle, 114, 0.338483], [bdyp, ruffle, 114, 0.422495], [acni,
     ruffle, 114, 0.449296], [hpet, ruffle, 114, 0.487456], [vgaa, ruffle, 114, 0.542811], [uuqr, ruffle, 114, 0.647225]])
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, 0.141351], [ybgx, ruffle, 114, 0.238483], [bdyp, ruffle, 114, 0.322495], [acni,
     ruffle, 114, 0.349296], [hpet, ruffle, 114, 0.387456], [vgaa, ruffle, 114, 0.442811], [uuqr, ruffle, 114, 0.547225]])
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, 0.041351], [ybgx, ruffle, 114, 0.138483], [bdyp, ruffle, 114, 0.222495], [acni,
     ruffle, 114, 0.249296], [hpet, ruffle, 114, 0.287456], [vgaa, ruffle, 114, 0.342811], [uuqr, ruffle, 114, 0.447225]])
ctm.obs: ctm.chunk.c.l([[jpne, ruffle, 114, -0.058649], [ybgx, ruffle, 114, 0.038483], [bdyp, ruffle, 114, 0.122495], [acni,
     ruffle, 114, 0.149296], [hpet, ruffle, 114, 0.187456], [vgaa, ruffle, 114, 0.242811], [uuqr, ruffle, 114, 0.347225]])
ctm.obs: ctm.chunk.d(jpne, ruffle, 114, -0.158649)
ctm.obs: ctm.chunk.c.l([[ybgx, ruffle, 114, -0.061517], [bdyp, ruffle, 114, 0.022495], [acni, ruffle, 114, 0.049296], [hpet,
     ruffle, 114, 0.087456], [vgaa, ruffle, 114, 0.142811], [uuqr, ruffle, 114, 0.247225]])
ctm.obs: ctm.chunk.d(ybgx, ruffle, 114, -0.161517)
ctm.obs: ctm.chunk.c.l([[bdyp, ruffle, 114, -0.077505], [acni, ruffle, 114, -0.050704], [hpet, ruffle, 114, -0.012544], [vgaa,
      ruffle, 114, 0.042811], [uuqr, ruffle, 114, 0.147225]])
ctm.obs: ctm.chunk.d(bdyp, ruffle, 114, -0.177505)
ctm.obs: ctm.chunk.d(acni, ruffle, 114, -0.150704)
ctm.obs: ctm.chunk.d(hpet, ruffle, 114, -0.112544)
ctm.obs: ctm.chunk.c.l([[vgaa, ruffle, 114, -0.057189], [uuqr, ruffle, 114, 0.047225]])
ctm.obs: ctm.chunk.d(vgaa, ruffle, 114, -0.157189)
ctm.obs: ctm.chunk.c.l([[uuqr, ruffle, 114, -0.052775]])
ctm.obs: ctm.chunk.d(uuqr, ruffle, 114, -0.152775)
?- #ctm.actor.list.store(:k,:l)
```

In the end, no *list* was memorized. Obviously this is a contrived example. If memorizing a *list* of numbers was important for the CTM, the weight associated with the *chunk* should have been set as high as needed to insure that the *list* will not have been dropped from the *stage*.

Adding numbers from a list consciously

In this example, we are going to assume that the list of numbers from the previous example was memorized and that we now need to calculate the sum of all the numbers *consciously*. This entail the need to keep the *list* on *stage* as well as the position within the *list* at which we currently are, and the value of the sum so far. We will also keep the length of the *list*. That is four different *chunks* that need to be on *stage* at the same time for each steps of the operation to be carried out. To help with this, we are going to add a support *actor* whose role will be to keep a set of *variables* and maintains them *on stage* whenever possible so that they can be used by any *actor*.

Create a new *fizz* file, calling it `variables.fizz`. The *actor* we are going to define in it will provide an interface for others to use in order to create new *variables*, change their values or delete them:

```
ctm.actor.variables {

    weight = 0.6,
    vars   = {}

} {

    // set the value of a variable identified by a label (new variable)
    (set,:label,:value) :-  peek(vars,:vars), !frm.label(:vars,:label)^,
                            set(%sym.4,:guid),
                            frm.store(:vars,:label,[:guid,:value],:vars.o),
                            poke(vars,:vars.o),
                            declare(ctm.chunk.u(:guid,variable,[:label,:value],$weight));

    // set the value of a variable identified by a label (replace value)
    (set,:label,:value) :-  peek(vars,:vars), frm.fetch(:vars,:label,[:guid,_]),
                            frm.store(:vars,:label,[:guid,:value],:vars.o),
                            poke(vars,:vars.o),
                            declare(ctm.chunk.u(:guid,variable,[:label,:value],$weight));

    // get the value stored for a given label (if the variable isn't yet on stage, the call won't complete)
    (get,:label,:value) :-  peek(vars,:vars),
                            frm.fetch(:vars,:label,[_,:value]);

    // remove an existing value using its label
    (cls,:label)        :-  peek(vars,:vars), frm.fetch(:vars,:label,[:guid,:value]),
                                frm.erase(:vars,:label,:vars.o),
                                poke(vars,:vars.o),
                                declare(ctm.chunk.u(:guid,variable,[:label,:value],$stage.drop));

    // regularly, we update out chunks to keep them on stage
    ()  :-  @ctm.tick.dull(_,_),
            frm.pairs($vars,:pairs),
            lst.member([:label,[:guid,:value]],:pairs),
            declare(ctm.chunk.u(:guid,variable,[:label,:value],$weight)),
            hush;

}
```

The *elemental* uses its `vars` property to store all the *variables* that will be created. The last defined *prototype* relies on one of the ticker to emit a *chunk* per *variables*. The *primitive* `frm.pairs` unifies its second *term* with a *list* containing each of the *variables*. It then uses the *primitive* `lst.member` to iterate over all of the *variables* stored in that list and emit a *chunk* for each one. This work because *fizz* will create as many concurrent inferring as needed to handle all the *statements* generated by the call to `lst.member`.

The first four *prototypes* don't show anything that we haven't seen so far in this article. For each, the *symbol* used as the first *term*, helps the *elemental* pick the right *prototype* to execute when a query is made. Whenever a *variable* is created or modified, the corresponding *chunk* will be emitted by the *elemental*. Note that we are creating an GUID for each of the *variables*. It will be used to identify the *chunk*.

To try this new *actor*, let's copy the solution file from the previous example and rename it `sumlist-c.fizz` and replace in it `memlist.fizz` by `variables.fizz`. We can then load the solution and use the console to instruct `ctm.actor.variables` to create a *conscious variable*, then modify its value and finally remove it:

```
$ ./fizz.x64 ./etc/experiments/ctm/sumlist-c.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/sumlist-c.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.007s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.002s
load : loading ./etc/experiments/ctm/variables.fizz ...
load : loaded ./etc/experiments/ctm/variables.fizz in 0.007s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.037s
load : loading completed in 0.040s
?- #ctm.actor.variables(set,a,12)
-> (  ) := 1.00 (0.002) 1
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.j(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.500000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.400000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.300000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.500000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.500000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.400000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.300000]])
?- #ctm.actor.variables(set,a,13)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.500000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 12], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.400000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 12], 0.600000]])
-> (  ) := 0.00 (0.001) 1
-> (  ) := 1.00 (0.002) 2
ctm.obs: ctm.chunk.u(wspf, variable, [a, 13], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.600000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 13], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.500000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.500000]])
?- #ctm.actor.variables(cls,a)
ctm.obs: ctm.chunk.u(wspf, variable, [a, 13], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.400000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.500000]])
ctm.obs: ctm.chunk.u(wspf, variable, [a, 13], 0.600000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.600000]])
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], 0.500000]])
-> (  ) := 1.00 (0.002) 1
ctm.obs: ctm.chunk.u(wspf, variable, [a, 13], -0.100000)
ctm.obs: ctm.chunk.c.l([[wspf, variable, [a, 13], -0.100000]])
ctm.obs: ctm.chunk.d(wspf, variable, [a, 13], -0.200000)
```

Note that removing a *variable* isn't immediate as there is no support in our implementation of a CTM for removing a *chunk*. What we do instead is to update the *chunk* with a weight that is the threshold used by the *stage* (the constant `$stage.drop`).

Let's now create a new *fizz* file (call it `sumlist-c.fizz`) to contain the rest of the example. We will start by adding an *elemental* to hold the *list* for which the sum must be computed:

```
ctm.actor.list.store {

    (xgib,[5,2,7,4,1]);

}
```

We are then going to add the *actor*, using two *elementals* to implement it. The main one, called `ctm.actor.sum`, will use a keypress event to start the computation and uses the *stage* frequent broadcasts as the way to step in the computation:

```
ctm.actor.sum {

    list.key = xgib

} {

    // when the 's' key is pressed, we get the list and creates the "variables"
    ()  :-  @console.keypress(115),
            #ctm.actor.list.store($list.key,:l),
            lst.length(:l,:s),
            #ctm.actor.variables(set,index,0),
            #ctm.actor.variables(set,count,:s),
            #ctm.actor.variables(set,sum,0),
            #ctm.actor.variables(set,list,:l),
            hush;

    // each the chunks on stage get broadcasted (as a frame), we get the variable and insure that we have them all and
     continue the computation
    ()  :-  @ctm.chunk.c.f({variable = :vars}), !is.variable(:vars),
            lst.member([_,[list,:l],_],:vars),
            lst.member([_,[index,:i],_],:vars),
            lst.member([_,[count,:n],_],:vars),
            lst.member([_,[sum,:s],_],:vars),
            #ctm.actor.sum.compute(:l,:n,:i,:s),
            hush;

}
```

When the key `S` is pressed, the *elemental* will query `ctm.actor.list.store` for the *list*, get its length and then set the four *variables* using `ctm.actor.variables`, each with an appropriate initial value. The second *prototype*, uses the broadcasting of the *stage* as a *frame* to get all the *variables* and retrieve the actual value of each ones in order to perform the computation. If *distractions* are causing some of the *variables* to be dropped from the *stage*, the corresponding *primitive* call to `lst.member` will fails causing the inferring to fails. Eventually, since the *actor* `ctm.actor.variables` will keep emitting the *variables* as *chunks*, there would come a time when all four *variables* will again be *conscious* at the same time. And thus the computation will resume.

The last bit of this example we need to define is the *elemental* `ctm.actor.sum.compute` which given all four *variables*, performs the addition of the number at the current index in the *list* and update the value of the appropriate *variables*:

```
ctm.actor.sum.compute { // compute the sum, step-by-step and update the variables on stage

    (:l,:n,:n,:s)^  :-  #ctm.actor.variables(cls,index),
                        #ctm.actor.variables(cls,list),
                        #ctm.actor.variables(cls,count),
                        #ctm.actor.variables(cls,sum),
                        console.puts("sum = ",:s);

    (:l,:n,:i,:s)   :-  lst.item(:l,:i,:v), add(:s,:v,:s2), add(:i,1,:i2),
                        #ctm.actor.variables(set,index,:i2),
                        #ctm.actor.variables(set,sum,:s2);

}
```

The first *prototype* handle the end of the operation. When the index in the list is equal (second and third *terms* are the same) to the length of the *list*, we will output the computed sum of the *list* and remove all four *variables*. For all the other times the *elemental* get queried, we retrieve (using `lst.item`) the number at the given index in the *list*, add its value to the current sum value and increase the index in the *list* by 1. The *prototype* ends by updating the value of the two *variables* that changed during the iteration over the *list*.

Once you have added the new file we just created to `sumlist-c.json`, you are ready to give this a try:

```
$ ./fizz.x64 ./etc/experiments/ctm/sumlist-c.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/sumlist-c.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.012s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.003s
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.002s
load : loading ./etc/experiments/ctm/variables.fizz ...
load : loaded ./etc/experiments/ctm/variables.fizz in 0.007s
load : loading ./etc/experiments/ctm/sumlist-c.fizz ...
load : loaded ./etc/experiments/ctm/sumlist-c.fizz in 0.008s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.042s
load : loading completed in 0.045s
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 0], 0.600000)
ctm.obs: ctm.chunk.u(ucba, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 0], 0.600000)
ctm.obs: ctm.chunk.u(gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.j(vpxi, variable, [index, 0], 0.600000)
ctm.obs: ctm.chunk.j(ucba, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.j(towl, variable, [sum, 0], 0.600000)
ctm.obs: ctm.chunk.j(gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[vpxi, variable, [index, 0], 0.600000], [ucba, variable, [count, 5], 0.600000], [towl, variable, [sum,
      0], 0.600000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000]])
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 5], 0.600000)
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.u(ucba, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 5], 0.600000)
ctm.obs: ctm.chunk.u(gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], 0.500000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.500000], [vpxi,
     variable, [index, 1], 0.600000], [towl, variable, [sum, 5], 0.600000]])
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 2], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 7], 0.600000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], 0.600000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000], [vpxi,
     variable, [index, 2], 0.600000], [towl, variable, [sum, 7], 0.600000]])
```

```
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 3], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 14], 0.600000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], 0.500000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.500000], [vpxi,
     variable, [index, 3], 0.600000], [towl, variable, [sum, 14], 0.600000]])
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 4], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 18], 0.600000)
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 4], 0.600000)
ctm.obs: ctm.chunk.u(ucba, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 18], 0.600000)
ctm.obs: ctm.chunk.u(gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], 0.400000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.400000], [vpxi,
     variable, [index, 4], 0.600000], [towl, variable, [sum, 18], 0.600000]])
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 5], 0.600000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 19], 0.600000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], 0.600000], [gbae, variable, [list, [5, 2, 7, 4, 1]], 0.600000], [vpxi,
     variable, [index, 5], 0.600000], [towl, variable, [sum, 19], 0.600000]])
sum = 19
ctm.obs: ctm.chunk.u(vpxi, variable, [index, 5], -0.100000)
ctm.obs: ctm.chunk.u(gbae, variable, [list, [5, 2, 7, 4, 1]], -0.100000)
ctm.obs: ctm.chunk.u(ucba, variable, [count, 5], -0.100000)
ctm.obs: ctm.chunk.u(towl, variable, [sum, 19], -0.100000)
ctm.obs: ctm.chunk.c.l([[ucba, variable, [count, 5], -0.100000], [gbae, variable, [list, [5, 2, 7, 4, 1]], -0.100000], [vpxi,
     variable, [index, 5], -0.100000], [towl, variable, [sum, 19], -0.100000]])
ctm.obs: ctm.chunk.d(ucba, variable, [count, 5], -0.200000)
ctm.obs: ctm.chunk.d(gbae, variable, [list, [5, 2, 7, 4, 1]], -0.200000)
ctm.obs: ctm.chunk.d(vpxi, variable, [index, 5], -0.200000)
ctm.obs: ctm.chunk.d(towl, variable, [sum, 19], -0.200000)
```

There we have it, the sum of the *list* is indeed `19`.

Enrolling help from others

Let's now look at variation of the previous example, where the knowledge of how to add two numbers is unknown to the *actor* `ctm.actor.sum.compute`. When faced with this situation, the *actor* must attempt to enroll the help of any *actor* capable of performing the task at hand.

To implement this, we will going to use a pattern that will rely on the *stage* for the broadcasting of a *query chunk* to all *actors*. When an *actor* gets a *query* that it can answers, it will push onto the *stage* a *reply chunk* which will contains the `GUID` of the *query*. This will allow the *actor* having posted the *query* to pick any corresponding *replies* from the *stage*. The template we will use for the *query chunks* will be `ctm.chunk.u(jjuu,query,add(5,2),1)` while the *reply chunks* will be expected to be akin to `ctm.chunk.u(nvqk,reply,[jjuu,add(5,2,7)],1)`.

To get started, make a copy of the *fizz* file called `sumlist-c.fizz` naming it `sumlist-c2.fizz`. We then will add a new *actor* that is capable of adding two numbers (by using the `add` primitive):

```
ctm.actor.binary.op {

    queries = {}

} {

    ()  :-  @ctm.chunk.j(:g,query,add(:a,:b),:w),
            !frm.label($queries,:g),
            set(%sym.4,:g.r),
            frm.store($queries,:g,:g.r,:q),
            poke(queries,:q),
            add(:a,:b,:c),
            declare(ctm.chunk.u(:g.r,reply,[:g,add(:a,:b,:c)],:w));

    ()  :-  @ctm.chunk.d(:g,query,add(_,_),_),
            frm.fetch($queries,:g,:g.r),
            frm.erase($queries,:g,:q),
            poke(queries,:q),
            declare(ctm.chunk.u(:g.r,reply,[:g,null],$stage.drop));

}
```

The *actor* contains two trigger based *prototypes* which will allow it to react to a *query chunk* being either added or removed to/from the *stage*. As the *actor* will reply by pushing onto the *stage* its own *chunk*. The *actor* is going to keep track of the *queries* it has answered so that it can cleanup the *reply chunk* from the *stage* when the original *query chunk* gets dropped from the *stage*. Note that it isn't mandatory as eventually, the *stage* will dropped the *chunk* as its weight trend down. Line 8 to 11 modifies the `queries` *property* of the *actor* to store a mapping of the *query's* `GUID` to the *reply's* `GUID`. On line 16 to 18, the mapping is tested then removed from the *property*. Both *prototypes* make uses of the `declare` *primitive* to publish the *reply chunk*, as we have seen in the previously examples.

Now, we need to modify the *actor* `ctm.actor.sum.compute` to send a *query chunk* and handle any *reply* instead of calling the `add` *primitive*.

```
 1  ctm.actor.sum.compute {
 2
 3      add.value = null,
 4      add.index = null
 5
 6  } {
 7
 8      (:l,:n,:n,:s)^  :-  #ctm.actor.variables(cls,index),
 9                          #ctm.actor.variables(cls,list),
10                          #ctm.actor.variables(cls,count),
11                          #ctm.actor.variables(cls,sum),
12                          console.puts("sum = ",:s);
13
14      (:l,:n,:i,:s)   :-  peek(add.value,null), peek(add.index,null),
15                          poke(add.value,%sym.4), poke(add.index,%sym.4),
16                          lst.item(:l,:i,:v),
17                          declare(ctm.chunk.u($add.value,query,add(:s,:v),1)),
18                          declare(ctm.chunk.u($add.index,query,add(:i,1),1));
19
20      ()              :-  @ctm.chunk.j(_,reply,[$add.value,add(_,_,:r)],_),
21                          poke(add.value,null),
22                          #ctm.actor.variables(set,sum,:r);
23
24      ()              :-  @ctm.chunk.j(_,reply,[$add.index,add(_,_,:r)],_),
25                          poke(add.index,null),
26                          #ctm.actor.variables(set,index,:r);
27
28  }
```

To do that, we modify the second *prototype* (on line 14) to declare two *query chunks*, one for adding the value at the current position in the *list* to the running sum, and the second one to increment the value of the *index*. We use two *properties* of the *actor* to store the `GUID` assigned to each of the *chunks* so that we can match any *reply* to the correct one. The last two *prototypes* are trigger based and allow the *actor* to react to any matching *reply chunks*. They basically do what the original version of the *actor* was doing, by setting the newly calculated value of the *variables* `sum` and `index`. In it, we also (line 21 and 25) reset the value of the *properties* to `null`. Without doing so, the above *prototype* will never complete since it checks that both *properties* are `null` (line 14).

Last, but not least, we need to make a small modification to the main *actor* `ctm.actor.sum`. Since the *actor* depends on the frequent broadcasting of the *stage* to perform each steps in the summation (one per broadcast), the usage of the *query/reply* pattern will cause the *actor* to no longer be in sync with the *stage's* broadcasts. We can fix that, by only moving forward in the computation steps when the value of the `index` variable is different from the previous one:

```
ctm.actor.sum {

    list.key = xgib,
    index    = -1

} {

    ()  :-  @console.keypress(115),
            #ctm.actor.list.store($list.key,:l),
            lst.length(:l,:s),
            #ctm.actor.variables(set,index,0),
            #ctm.actor.variables(set,count,:s),
            #ctm.actor.variables(set,sum,0),
            #ctm.actor.variables(set,list,:l),
            poke(index,-1),
            hush;

    ()  :-  @ctm.chunk.c.f({variable = :vars}), !is.variable(:vars),
            lst.member([_,[list,:l],_],:vars),
            lst.member([_,[index,:i?[neq($index)]],_],:vars),
            lst.member([_,[count,:n],_],:vars),
            lst.member([_,[sum,:s],_],:vars),
            poke(index,:i),
            #ctm.actor.sum.compute(:l,:n,:i,:s),
            hush;

}
```

As we have seen before, the first *prototype* use the `'s'` key trigger to get the list then creates the "variables". The second *prototype* also works as before, computing the sum of the list based on the *variables*' values. This time however, like we discussed, we check that the `index` has really changed since the last broadcasting from the *stage*.

To test this, copy the `JSON` file called `sumlist-c.json` into `sumlist-c2.json` then modify its fourth line to point to the new *fizz* file we just created. You can then run it like we did before. You will notice that this time around, because there's more broadcasting from the *stage* in between summation steps, it takes sensibly longer to compute the total sum.

Making connections at runtime

An inportant part of the *CTM* concept is that learning in implemented (in part) by having actors directly talking to other actors instead of going via the *stage*. Originaly, an *actor* doesn't know of any others and so when it does need to get something done or need help figuring something up, it relies on the *stage* broadcasting to reach anybody that can be of help. This is what we saw in the above revised example. Overtime, if the answers delivered by an actor to queries from another actor are deemed valuable, a direct link between the two actors can be established, forgoing the *stage*. In this section, wer are going to modify the previous example to demonstrate that point. Note that this (like all other examples, in fact) is a much simplified scenario. As such, we will assume that the *actor* tasked with computing the sum of the list will create a link with the very first *actor* to send a reply to its *"add two numbers"* query.

To get started, we will make a copy of the *fizz* file called `sumlist-c2.fizz` naming it `sumlist-c3.fizz`. We then modify the *actor* `ctm.actor.binary.op` by adding to it a few *prototypes* that can be called directly to compute, for example, the sum of two numbers in line 21 to 24:

```
 1 ctm.actor.binary.op {
 2
 3     queries = {}
 4
 5 } {
 6
 7     ()  :-  @ctm.chunk.j(:g,query,add(:a,:b),:w),
 8             !frm.label($queries,:g),
 9             set(%sym.4,:g.r),
10             frm.store($queries,:g,:g.r,:q),
11             poke(queries,:q),
12             add(:a,:b,:c),
13             declare(ctm.chunk.u(:g.r,reply,[:g,$self,add(:a,:b,:c)],:w));
14
15     ()  :-  @ctm.chunk.d(:g,query,add(_,_),_),
16             frm.fetch($queries,:g,:g.r),
17             frm.erase($queries,:g,:q),
18             poke(queries,:q),
19            declare(ctm.chunk.u(:g.r,reply,[:g,$self,null],$stage.drop));
20
21     (add(:a,:b),:c)  :- add(:a,:b,:c);
22     (sub(:a,:b),:c)  :- sub(:a,:b,:c);
23     (mul(:a,:b),:c)  :- mul(:a,:b,:c);
24     (div(:a,:b),:c)  :- div(:a,:b,:c);
25
26 }
```

Another change that we made is to add the label of the *actor* it-self to the *chunks* that the *actor* delares (line 13 and 19) using the *constant* `$self` as the second *term* in the data contained by the *chunks*. This will enable any *actor* to know from whom the reply came from and use that to directly query the *actor*.

Next, we will modify the `ctm.actor.sum.compute` actor to allow it to notice who replied to its query so that it can directly query it for any further summation steps. Let's start by adding the following *prototype*:

```
1     (:l,:n,:i,:s)   :-  peek(helper,_?[neq(null)])^,
2                         lst.item(:l,:i,:v),
3                         #ctm.actor.sum.compute.add(:s,:v,:s2),
4                         #ctm.actor.sum.compute.add(:i,1,:i2),
5                         #ctm.actor.variables(set,index,:i2),
6                         #ctm.actor.variables(set,sum,:s2);
```

It offers an alternative to be considered by the solver when `ctm.actor.sum` is querying the *actor* at each *stage* broadcast. For the *actor* to know when to post its query to the *stage* or to go the direct route, we will use a new *property* which we will call `helper`. On line 1, we fetch (using the *primitive* `peek`) the value of that *property* and insure that the inference continues only if the value doesn't unifies with the symbol `null` (which we will have to initialize the *property* to). On line 3 and 4, we now have a predicate to an elemental called `ctm.actor.sum.compute.add`. This elemental will be the one that will link adding two numbers with the *actor* that knows how to do it.

We now modify the original *prototype* to include a check that the very same *property* unifies to `null`. When the *elemental* is getting queried by the *actor* `ctm.actor.sum`, both of the *prototypes* will be considered but only one will be selected for further inferencing based on the value of the `helper` property:

```
1     (:l,:n,:i,:s)   :-  peek(helper,null)^,
2                         peek(add.value,null), peek(add.index,null),
3                         poke(add.value,%sym.4), poke(add.index,%sym.4),
4                         lst.item(:l,:i,:v),
5                         declare(ctm.chunk.u($add.value,query,add(:s,:v),1)),
6                         declare(ctm.chunk.u($add.index,query,add(:i,1),1));
```

The two trigger based *prototypes* also need to be modified to take into consideration the added *term* to the *chunks* `ctm.chunk.j`:

```
1     ()              :-  @ctm.chunk.j(_,reply,[$add.value,:s,add(_,_,:r)],_),
2                         poke(add.value,null),
3                         #ctm.actor.variables(set,sum,:r),
4                         ~self(link(:s));
5
6     ()              :-  @ctm.chunk.j(_,reply,[$add.index,:s,add(_,_,:r)],_),
7                         poke(add.index,null),
8                         #ctm.actor.variables(set,index,:r),
9                         ~self(link(:s));
```

At the end of both *prototypes*, we also add a recursive *predicate* passing to it for *term* a *functor* that contains the *label* of the *actor* that replied. This will be implemented in the *elemental* as follow:

```
    (link(:s))      :-  peek(helper,null)^, poke(helper,:s),
                        sym.cat($self,".add",:l),
                        define(:l,[\:a,\:b,\:c],[],[
                            [[],[:s,[add(\:a,\:b),\:c]]
                        ]);
    (link(_))       :-  true;
```

In the first *prototype*, we first insure that the `helper` value is still `null`, then change it to the label of the *actor*. On line 2, we create a new *symbol* by concatenating the label of the *elemental* with `".add"` before calling the *primitive* `define` to specify a *prototype* to be created. If there is no existing *elemental* with that label, the *substrate* will instantiate one, otherwise the new *prototype* will be inserted in the *knowledge* of an *elemental* named `ctm.actor.sum.compute.add`. Thus, from that point on, the *actor* will directly query the *actor* instead of posting on the *stage*. The second *prototype* is necessary to insure that such query always returns successfully when the `helper` property has already been set, that is every summation step after the firt one.

Let's give this new example a try:

```
$ ./fizz.x64 ./etc/experiments/ctm/sumlist-c3.json
fizz 0.6.0-D (20190502.2241) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/sumlist-c3.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loading ./etc/experiments/ctm/ruffle.fizz ...
load : loaded ./etc/experiments/ctm/ruffle.fizz in 0.002s
load : loading ./etc/experiments/ctm/variables.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.002s
load : loading ./etc/experiments/ctm/sumlist-c3.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.007s
load : loaded ./etc/experiments/ctm/variables.fizz in 0.007s
load : loaded ./etc/experiments/ctm/sumlist-c3.fizz in 0.019s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.032s
load : loading completed in 0.035s
ctm.obs: ctm.chunk.u(qdap, variable, [index, 0], 0.600000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 0], 0.600000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.u(qdap, variable, [index, 0], 0.600000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 0], 0.600000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.j(qdap, variable, [index, 0], 0.600000)
ctm.obs: ctm.chunk.j(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.j(txlc, variable, [sum, 0], 0.600000)
ctm.obs: ctm.chunk.j(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[qdap, variable, [index, 0], 0.600000], [duuf, variable, [count, 5], 0.600000], [txlc, variable, [sum,
      0], 0.600000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000]])
ctm.obs: ctm.chunk.u(umqn, query, add(0, 5), 1)
ctm.obs: ctm.chunk.u(tvwq, query, add(0, 1), 1)
```

As before, the `ctm.actor.sum` actor establishes on the *stage* as many *chunks* as it needs *variables* to *consciously* performs the operation requested from it. It then goes on to pushing onto the stage two *query chunks*, the first one to add the value `5` to the current value of the sum, and the second to increase the index within the `list` by `1`. As both *chunks* get accepted onto the *stage*, they are eventually seen by the *actor* `ctm.actor.binary.op`:

```
ctm.obs: ctm.chunk.j(umqn, query, add(0, 5), 1)
ctm.obs: ctm.chunk.u(jvym, reply, [umqn, ctm.actor.binary.op, add(0, 5, 5)], 1)
ctm.obs: ctm.chunk.j(tvwq, query, add(0, 1), 1)
ctm.obs: ctm.chunk.u(snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 1)
```

The *actor* performs the requested computation, then push onto the *stage* a *reply chunk*, identifying the *query* it is replying too, as well as identifying itself.

```
ctm.obs: ctm.chunk.c.l([[qdap, variable, [index, 0], 0.600000], [duuf, variable, [count, 5], 0.600000], [txlc, variable, [sum,
      0], 0.600000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000], [umqn, query, add(0, 5), 1], [tvwq, query, add(0,
     1), 1]])
ctm.obs: ctm.chunk.d(qdap, variable, [index, 0], 0.500000)
ctm.obs: ctm.chunk.j(jvym, reply, [umqn, ctm.actor.binary.op, add(0, 5, 5)], 1)
ctm.obs: ctm.chunk.j(snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 1)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 5], 0.600000)
ctm.obs: ctm.chunk.u(qdap, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.c.l([[duuf, variable, [count, 5], 0.500000], [txlc, variable, [sum, 0], 0.500000], [vpcf, variable, [list,
     [5, 2, 7, 4, 1]], 0.500000], [umqn, query, add(0, 5), 0.900000], [tvwq, query, add(0, 1), 0.900000], [jvym, reply, [umqn
     , ctm.actor.binary.op, add(0, 5, 5)], 1], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 1]])
```

When the *reply chunks* are noticed by the *actor* `ctm.actor.sum`, the related *variables* are updated by pushing onto the *stage* the updated *chunk*:

```
ctm.obs: ctm.chunk.u(qdap, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 5], 0.600000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.d(duuf, variable, [count, 5], 0.400000)
ctm.obs: ctm.chunk.j(qdap, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.c.l([[vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.400000], [txlc, variable, [sum, 5], 0.600000], [qdap,
     variable, [index, 1], 0.600000], [umqn, query, add(0, 5), 0.800000], [tvwq, query, add(0, 1), 0.800000], [jvym, reply, [
     umqn, ctm.actor.binary.op, add(0, 5, 5)], 0.900000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)],
     0.900000]])
```

As the capacity of the *stage* is limited, some of the *chunk* may get dropped as we see above and below. While this may results in delay, the system is resilient as the *actor* will continue pushing its *variables chunks* at each *tick*.

```
ctm.obs: ctm.chunk.d(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.j(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.c.l([[txlc, variable, [sum, 5], 0.600000], [qdap, variable, [index, 1], 0.600000], [duuf, variable, [count,
      5], 0.600000], [umqn, query, add(0, 5), 0.700000], [tvwq, query, add(0, 1), 0.700000], [jvym, reply, [umqn, ctm.actor.
     binary.op, add(0, 5, 5)], 0.800000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.800000]])
ctm.obs: ctm.chunk.c.l([[txlc, variable, [sum, 5], 0.500000], [qdap, variable, [index, 1], 0.500000], [duuf, variable, [count,
      5], 0.500000], [umqn, query, add(0, 5), 0.600000], [tvwq, query, add(0, 1), 0.600000], [jvym, reply, [umqn, ctm.actor.
     binary.op, add(0, 5, 5)], 0.700000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.700000]])
ctm.obs: ctm.chunk.u(qdap, variable, [index, 1], 0.600000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 5], 0.600000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[txlc, variable, [sum, 5], 0.400000], [qdap, variable, [index, 1], 0.400000], [duuf, variable, [count,
      5], 0.400000], [umqn, query, add(0, 5), 0.500000], [tvwq, query, add(0, 1), 0.500000], [jvym, reply, [umqn, ctm.actor.
     binary.op, add(0, 5, 5)], 0.600000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.600000]])
ctm.obs: ctm.chunk.d(umqn, query, add(0, 5), 0.400000)
ctm.obs: ctm.chunk.u(jvym, reply, [umqn, ctm.actor.binary.op, null], -0.100000)
ctm.obs: ctm.chunk.j(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.c.l([[tvwq, query, add(0, 1), 0.400000], [jvym, reply, [umqn, ctm.actor.binary.op, add(0, 5, 5)],
     0.500000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.500000], [txlc, variable, [sum, 5], 0.600000], [
     qdap, variable, [index, 1], 0.600000], [duuf, variable, [count, 5], 0.600000], [vpcf, variable, [list, [5, 2, 7, 4, 1]],
      0.600000]])
```

Eventually though, the *actor* can continue perfoming the steps of its computation, this time by using the direct link it created to the *actor* `ctm.actor.binary.op` as we can observe below by the lack of *query chunks*:

```
ctm.obs: ctm.chunk.u(qdap, variable, [index, 2], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 7], 0.600000)
ctm.obs: ctm.chunk.c.l([[jvym, reply, [umqn, ctm.actor.binary.op, null], -0.100000], [tvwq, query, add(0, 1), 0.300000], [snsc
     , reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.400000], [duuf, variable, [count, 5], 0.500000], [vpcf, variable,
     [list, [5, 2, 7, 4, 1]], 0.500000], [txlc, variable, [sum, 7], 0.600000], [qdap, variable, [index, 2], 0.600000]])
ctm.obs: ctm.chunk.u(qdap, variable, [index, 3], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 14], 0.600000)
ctm.obs: ctm.chunk.u(qdap, variable, [index, 3], 0.600000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 14], 0.600000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000)
ctm.obs: ctm.chunk.d(jvym, reply, [umqn, ctm.actor.binary.op, null], -0.200000)
ctm.obs: ctm.chunk.c.l([[tvwq, query, add(0, 1), 0.200000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)],
     0.300000], [duuf, variable, [count, 5], 0.400000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.400000], [txlc, variable,
      [sum, 14], 0.600000], [qdap, variable, [index, 3], 0.600000]])
```

```
ctm.obs: ctm.chunk.u(qdap, variable, [index, 4], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 18], 0.600000)
ctm.obs: ctm.chunk.c.l([[tvwq, query, add(0, 1), 0.100000], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)],
     0.200000], [duuf, variable, [count, 5], 0.600000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.600000], [txlc, variable,
      [sum, 18], 0.600000], [qdap, variable, [index, 4], 0.600000]])
ctm.obs: ctm.chunk.u(qdap, variable, [index, 5], 0.600000)
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 19], 0.600000)
ctm.obs: ctm.chunk.c.l([[tvwq, query, add(0, 1), 0], [snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], 0.100000], [duuf
     , variable, [count, 5], 0.500000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], 0.500000], [txlc, variable, [sum, 19],
     0.600000], [qdap, variable, [index, 5], 0.600000]])
ctm.obs: ctm.chunk.u(qdap, variable, [index, 5], -0.100000)
ctm.obs: ctm.chunk.u(vpcf, variable, [list, [5, 2, 7, 4, 1]], -0.100000)
ctm.obs: ctm.chunk.u(duuf, variable, [count, 5], -0.100000)
sum = 19
```

A few *ticks* later, the *actor* complete the summation of the `list` and cleanup of the *stage* occurs:

```
ctm.obs: ctm.chunk.u(txlc, variable, [sum, 19], -0.100000)
ctm.obs: ctm.chunk.d(tvwq, query, add(0, 1), -0.100000)
ctm.obs: ctm.chunk.u(snsc, reply, [tvwq, ctm.actor.binary.op, null], -0.100000)
ctm.obs: ctm.chunk.c.l([[duuf, variable, [count, 5], -0.100000], [vpcf, variable, [list, [5, 2, 7, 4, 1]], -0.100000], [txlc,
     variable, [sum, 19], -0.100000], [qdap, variable, [index, 5], -0.100000], [snsc, reply, [tvwq, ctm.actor.binary.op, add
     (0, 1, 1)], 0]])
ctm.obs: ctm.chunk.d(duuf, variable, [count, 5], -0.200000)
ctm.obs: ctm.chunk.d(vpcf, variable, [list, [5, 2, 7, 4, 1]], -0.200000)
ctm.obs: ctm.chunk.d(txlc, variable, [sum, 19], -0.200000)
ctm.obs: ctm.chunk.d(qdap, variable, [index, 5], -0.200000)
ctm.obs: ctm.chunk.d(snsc, reply, [tvwq, ctm.actor.binary.op, add(0, 1, 1)], -0.100000)
ctm.obs: ctm.chunk.j(snsc, reply, [tvwq, ctm.actor.binary.op, null], -0.100000)
ctm.obs: ctm.chunk.c.l([[snsc, reply, [tvwq, ctm.actor.binary.op, null], -0.100000]])
ctm.obs: ctm.chunk.d(snsc, reply, [tvwq, ctm.actor.binary.op, null], -0.200000)
```

Integrating in the up-tree

For the final example in this article, we are going to look at a component of the Blums' CTM concept which we have been ignoring so far: the *up-tree* that integrates related *chunks* before they are seen by the *stage*. In it, we will be looking at two *actors* attempting to push *chunks* that are in opposition onto the *stage*: one expressing a *positive* sentiment while the other expresses a *negative* sentiment.

Refering to the description of the CTM at the beginning of this article, we know that *chunks* can gets integrated between many *actors* that are (somehow) associated. While how the association is done is still unknown, the rule when this happens is that the *chunk* with the highest magnitude is the one getting pushed-up, with the weight as the sum of the weights of the *chunks*. For practical reason, we are going to assume here that the integration may happens between more than two *actors*. We will also assume that a new *chunk*, independent of the *chunks* involved in the integration is emitted as the integration occurs. Lastly, our implementation of the *up-tree* will see each node in the tree as an independent *elemental*.

To start, create a new *fizz* file calling it `utree.fizz`. In it, we are first going to add the two *actors* which will be linked to the same node on the *up-tree*. Since they will operate in the same way, we are going to make use of the *elemental* cloning support in *fizz* and only fully define the first one. Whenever the key associated with the *actor* is pressed, a *chunk* will be emitted. The more we emit the *chunk*, the higher the weight of the *chunk* will be:

```
ctm.actor.yay {

    key        = 121, // 'y' key
    label      = yay,
    weight     = 0,
    weight.rng = <0|1>
    loss       = -0.1,
    gain       = +0.1

} {

    // increase the weight at each keypress and send chunk
    ()  :-  @console.keypress($key),
            add($weight,$gain,:w.o),
            rng.clamp($weight.rng,:w.o,:w.c),
            poke(weight,:w.c),
            ~self(self,:u),
            declare(ctm.chunk.r(:u,$label,$key,:w.c)),
            hush;

    // lower the weight as time flow and send chunk
    ()  :-  @ctm.tick.slow(_,_),
            add($weight,$loss,:w.o),
            rng.clamp($weight.rng,:w.o,:w.c),
            poke(weight,:w.c),
            ~self(self,:u),
            declare(ctm.chunk.r(:u,$label,$key,:w.c)),
            hush;

    // get randomly assigned uid or create one if needed
    (self,:u) :- peek(uid,:u)^;
    (self,:u) :- set(:u,%sym.4), poke(uid,:u);

}

ctm.actor.nay {

    clone       = ctm.actor.yay,
    key         = 110, // 'n' key
    label       = nay,
    weight.rng  = <-1,0>
    loss        = +0.1,
    gain        = -0.1

} {}
```

By using different value for the same properties, we have setup `ctm.actor.yay` and `ctm.actor.nay` to work identically but in opposite. For example, the more we press in the `N` key, the lower the weight of the *chunk* will get for `ctm.actor.nay`. In order to work, we are going to have to use a solution file where the *stage*'s threshold value is much lower than what we have used before. Before we get any further, create the solution file `utree.json` and copy into it the following:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   ["stage.fizz", "debug.fizz", "ticks.fizz", "utree.fizz" ],
        "globals" :   [
            {
                "label" : "tick.hush",
                "value" : 5
            },
            {
                "label" : "tick.dull",
                "value" : 3
            },
            {
                "label" : "tick.slow",
                "value" : 1
            },
            {
                "label" : "tick.fast",
                "value" : 0.5
            },
            {
                "label" : "stage.size",
                "value" : 7
            },
            {
                "label" : "stage.loss",
                "value" : 0.1
            },
            {
                "label" : "stage.drop",
                "value" : -2
            },
            {
                "label" : "ruffle.weight.min",
                "value" : 0.7
            },
            {
                "label" : "ruffle.weight.max",
                "value" : 1.0
            }
        ]
    }
}
```

This set the threshold to the value of `-2` which will work better in this scenario. Now, you may have noticed in the *procedural knowledge* for `ctm.actor.yay` that instead of declaring a *chunk* with `ctm.chunk.u` we are using `ctm.chunk.r`. This is requiered since we want the *chunks* emitted by both of these *actors* to not go directly to the *stage*. The *elemental* we are creating to serve as the *up-tree* node will be using a *trigger predicate* for `ctm.chunk.r` and emitting its *chunks* with `ctm.chunk.u` if it is a final node or `ctm.chunk.r` if it is itself connected to another up-tree node.

Let's define now the *up-tree* node. We will rely on properties to configure the node as this will allow for the *elemental* to be easily cloned when multiple nodes are requiered:

```
ctm.utree.node {

    uid = xgib,
    frm = {},
    lbl = [yay,nay],
    out = ctm.chunk.u

} {

    () :-   @ctm.chunk.r(:g?[neq($uid)],:l?[lst.member($lbl)],:d,:w),
            frm.store($frm,:g,[:l,:d,:w],:frm.o), poke(frm,:frm.o),
            #ctm.utree.push($uid,:frm.o,$out),
            hush;

}
```

The node's properties are: the GUID to be assigned to any *chunk* emitted by the node (`uid` property), a frame containing all the *chunks* received (`frm` property), the labels of the *chunks* the node will accept (`lbl` property) and the type of *chunk* to send *up-tree* (`out` property).

The *elemental* contains a single *prototype* which uses, as discussed earlier, `ctm.chunk.r` as a *trigger predicate*. To minimize the false triggers, the first *term* in the *predicate* is constrained to not be the GUID employed by the *elemental*. This will matter when the node is connected to another node instead of sending its *chunks* directly to the *stage*. Also, since there could be more than one node dealing with *chunks*, we contraint the label of the *chunk* to be present in the `lbl` property. Assuming the inferring continues, the received *chunk* is added to the *frame* and we write it back onto the property before using a *secondary elemental* to carry out the integration:

```
ctm.utree.push {

    (:u,:f,:out)    :-  frm.labels(:f,:k.f), lst.mix(:k.f,:k),
                        #ctm.utree.select(:f,:k,[_,:l,:d,_],:w),
                        declare(:out,[:u,:l,:d,:w]);

}
```

The *elemental*'s *prototype* get a list of all the *chunks*' GUID in the *frame* (using the *property* `frm.labels`) then scramble the list using `lst.mix`. This may seems like an odd thing to do, but this will allow for the *chunks* emitted by the *elemental* to alternate between all the received *chunks* when they have the same weight. The inferring then use another *secondary elemental* to select the *chunk* and weight before it emit it. Since we will only care about the label and the data from the *chunk*, we unify the *list* that will be provided as third *term* to the *predicate* with a list where the head and tail *terms* are *wildcard variables*.

Selecting the right *chunk* means that we need to compute the sum of the weights of all the *chunks*, but that we also need to get the *chunks* with the highest magnitude. This is done here by using two *elementals*:

```
ctm.utree.select {

    (:f,:k,:c,:w) :- #ctm.utree.sum(:f,:k,:w), #ctm.utree.max.w(:f,:k,:c);

}
```

Here are the two new *elementals* definitions as well as one supporting *elemental*:

```
ctm.utree.max.a { // pick the chunk of two with the highest weight magnitude

    ([:k.1,:l.1,:d.1,:w.1],[:k.2,:l.2,:d.2,:w.2],[:k.1,:l.1,:d.1,:w.1]) :-  mao.abs(:w.1,:a.1), mao.abs(:w.2,:a.2),
                                                                          gte(:a.1,:a.2)^;
    ([:k.1,:l.1,:d.1,:w.1],[:k.2,:l.2,:d.2,:w.2],[:k.2,:l.2,:d.2,:w.2]) :-  true;

}

ctm.utree.max.w { // get the chunk stored in the frame with the highest weight magnitude

    (_,[],nil)^              :-  true;
    (:f,[:k],[:k,:l,:d,:w]) :-  frm.fetch(:f,:k,[:l,:d,:w]);
    (:f,[:k.1|:r],:c)       :-  frm.fetch(:f,:k.1,[:l.1,:d.1,:w.1]), ~self(:f,:r,[:k.2,:l.2,:d.2,:w.2]),
                                #ctm.utree.max.a([k.1,:l.1,:d.1,:w.1],[:k.2,:l.2,:d.2,:w.2],:c);

}

ctm.utree.sum { // compute the sum of the weights of all the chunks in the frame

    (_,[],0)^       :-  true;
    (:f,[:k],:w)    :-  frm.fetch(:f,:k,[_,_,:w]);
    (:f,[:k|:r],:s) :-  frm.fetch(:f,:k,[_,_,:w]), ~self(:f,:r,:s.1), add(:w,:s.1,:s);

}
```

If we run the example without changing anything to the *actors*, we can observe the flipping of the *chunk* coming from the *up-tree*:

```
$ ./fizz.x64 ./etc/experiments/ctm/utree.json
fizz 0.5.D-X (20181110.2324) [lnx.x64|4|w|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ctm/utree.json ...
load : loading ./etc/experiments/ctm/stage.fizz ...
load : loading ./etc/experiments/ctm/debug.fizz ...
load : loaded ./etc/experiments/ctm/debug.fizz in 0.009s
load : loading ./etc/experiments/ctm/ticks.fizz ...
load : loaded ./etc/experiments/ctm/ticks.fizz in 0.001s
load : loading ./etc/experiments/ctm/utree.fizz ...
load : loaded ./etc/experiments/ctm/utree.fizz in 0.011s
load : loaded ./etc/experiments/ctm/stage.fizz in 0.024s
load : loading completed in 0.028s
ctm.obs: ctm.chunk.r(rogm, yay, 121, 0)
ctm.obs: ctm.chunk.r(udiy, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.r(rogm, yay, 121, 0)
ctm.obs: ctm.chunk.r(udiy, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.j(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.c.l([[xgib, nay, 110, 0]])
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.r(rogm, yay, 121, 0)
ctm.obs: ctm.chunk.r(udiy, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.c.l([[xgib, nay, 110, 0]])
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.r(rogm, yay, 121, 0)
ctm.obs: ctm.chunk.r(udiy, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.u(xgib, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.c.l([[xgib, yay, 121, 0]])
```

If we now press the `Y` key a couple of times, we can see the *chunk* coming from the node is definitly positive:

```
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.300000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.300000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.300000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.200000)
ctm.obs: ctm.chunk.r(etve, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.c.l([[xgib, yay, 121, 0.300000]])
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.300000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.300000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.300000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.400000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.500000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.500000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.500000)
ctm.obs: ctm.chunk.r(dblg, yay, 121, 0.400000)
ctm.obs: ctm.chunk.r(etve, nay, 110, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.400000)
ctm.obs: ctm.chunk.c.l([[xgib, yay, 121, 0.500000]])
```

And we press the `N` key, we will observe the weight of the *chunk* drop as it becomes the sum of the weight of both *chunks*:

```
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.300000)
ctm.obs: ctm.chunk.r(xtom, nay, 110, -0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.200000)
ctm.obs: ctm.chunk.r(xtom, nay, 110, -0.200000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
ctm.obs: ctm.chunk.r(ocnh, yay, 121, 0.200000)
ctm.obs: ctm.chunk.r(xtom, nay, 110, -0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.c.l([[xgib, yay, 121, 0.100000]])
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0)
ctm.obs: ctm.chunk.u(xgib, yay, 121, 0.100000)
```