Semantic code transformations in MetaJS

Dmytro V. Dogadailo
www.coect.net
October, 2013

1 What is MetaJS?

1. Write on Lisp.
2. Compile to pure JavaScript.
3. Can generate missed parts of code.
4. New approach to literate programming.
5. Try in the browser: http://metajs.coect.net/

If a tree falls in a forest

and no one is around to hear it,

does it make a sound?

http://en.wikipedia.org/wiki/If_a_tree_falls_in_a_forest

(def user)

If a symbol has name “user”

and no declared type,

does it have properties of

the entity “user”?

2 Symbolic transformations

1. Name → Name
2. Name → Meta
3. Meta → Name
4. Meta → Meta

2.1 Name → Name

In the bellow example required argument **username** of the function **get-user-timeline** is missed in the function call. To fix this function call MetaJS uses local variable **username** because it's have same name as missed function argument and it's the only possible name resolution.

On the right is JavaScript code generated from MetaJS code. Resolved name highlighted with **red** colour. You can check this and next examples right inside your browser on the http://metajs.coect.net/.

```
(defn get-user-timeline (username)
      #"$username timeline")


(let* (username "dogada")
      (get-user-timeline))
```

```
var getUserTimeline = (function(username) {
  return ("" + username + " timeline");
});

(function(username) {
  return getUserTimeline(username);
})("dogada");
```

If there are several possible ways of missed symbol resolution, MetaJS **will not guess** and will issue a warning to the programmer. Programmer adds missed argument manually or refactors code into smaller blocks. The same approach is used in the version control systems. Most of time merges of different code branches are done automatically, but if there are a merge conflict – programmer need to resolve it manually or use different merge algorithm.

2.2 Name → Meta

Required argument **username** of the function **get-user-timeline** is missed in the function call. MetaJS uses local variable **u**, because it's declared with meta type **username**.

```
(defn get-user-timeline (username)
      #"$username timeline")


(let* (u:username "dogada")
      (get-user-timeline))
```

```
var getUserTimeline = (function(username) {
  return ("" + username + " timeline");
});

(function(u) {
  return getUserTimeline(u);
})("dogada");
```

2.3 Meta → Name

Required argument **user-id** of the function **get-user-timeline** is missed in the function call. MetaJS uses local variable **username**, because the function argument **user-id** is declared with meta type **username**.

```
(defn get-user-timeline (user-id:username)
      #"$user-id timeline")


(let* (username "dogada")
      (get-user-timeline))
```

```
var getUserTimeline = (function(userId) {
  return ("" + userId + " timeline");
});

(function(username) {
  return getUserTimeline(username);
})("dogada");
```

2.4 Meta → Meta

Required argument **user-id** of the function **get-user-id-timeline** is missed in the function call. MetaJS uses local variable **u**, because it and the function argument **user-id** is declared with same meta type **username**.

```
(defn get-user-timeline (user-id:username)
      #"$user-id timeline")


(let* (u:username "dogada")
      (get-user-timeline))
```

```
var getUserTimeline = (function(userId) {
  return ("" + userId + " timeline");
});

(function(u) {
  return getUserTimeline(u);
})("dogada");
```

3 Entitative transformations

1. Name → Entity → Name
2. Meta → Entity → Name
3. Name → Entity → Meta
4. Meta → Entity → Meta

3.1 Name → Entity → Name

Required argument **username** of the function **get-user-timeline** is missed in the function call. MetaJS uses property **username** of local variable **request**, because name of the property is identical to the name of the missed argument and there are no other possible candidates. MetaJS assumes that local variable **request** has property **username**, because local variable **request** implicitly associated with entity **request**.

```
(entity request
        "Entity of demo web request."
        (has username))

(defn get-user-timeline (username)
      #"$username timeline")

(let* (request {username: "dogada"})
      (get-user-timeline))
```

```
var getUserTimeline = (function(username) {
  return ("" + username + " timeline");
});

(function(request) {
  return getUserTimeline(request.username);
})({username: "dogada"});
```

Entities are macros and aren't translated to JavaScript code. Compiler uses entities to understand how to use symbols (variables and function arguments) associated with entities. Such association can be implicit (when symbol and entity have same name) and explict (for example, **req:request**). Entities are not classical types. Entities explains to the compiler meaning of the words used in your code.

3.2 Meta → Entity → Name

Required argument **user-id** of the function **get-user-timeline** is missed in the function call. MetaJS uses property **username** of local variable **request**, because entity **request** declares relation between **request** and **username** entities and name of the property is identical to the meta type (entity) of the missed argument.

```
(entity request
        "Entity of demo web request."
        (has username))

(defn get-user-timeline (user-id:username)
  #"$user-id timeline")

(let* (request {username: "dogada"})
      (get-user-timeline))
```

```
var getUserTimeline = (function(userId) {
  return ("" + userId + " timeline");
});

(function(request) {
  return getUserTimeline(request.username);
})({username: "dogada"});
```

3.3 Name → Entity → Meta

Required argument **username** of the function **get-user-timeline** is missed in the function call. MetaJS uses property **username** of local variable **req**, because it's declared with meta type (entity) **request** and entity **request** declares relation between **request** and **username** entities.

```
(entity request
        "Entity of demo web request."
        (has username))
```

```
(defn get-user-timeline (username)
  #"$username timeline")
```

```
(let* (req:request {username: "dogada"})
     (get-user-timeline))
```

```
var getUserTimeline = (function(username) {
  return ("" + username + " timeline");
});
```

```
(function(req) {
  return getUserTimeline(req.username);
})({username: "dogada"});
```

3.4 Meta → Entity → Meta

Required argument **user-id** of the function **get-user-timeline** is missed in the function call. MetaJS uses property **username** of local variable **req**, because it's declared with meta type (entity) **request** and entity **request** declares relation between **request** and **username** entities and **user-id** argument also declared as **username**.

```
(entity request
        "Entity of demo web request."
        (has username))

(defn get-user-timeline (user-id:username)
      #"$user-id timeline")

(let* (req:request {username: "dogada"})
      (get-user-timeline))
```

```
var getUserTimeline = (function(userId) {
  return ("" + userId + " timeline");
});

(function(req) {
  return getUserTimeline(req.username);
})({username: "dogada"});
```

How deep
does
the rabbit hole
go?

4 Complex transformations

- For transformation of single function call can be used several symbolic and entitative transformation.
- MetaJS resolves only required function arguments that is missed in the function call.
- For each missed symbol MetaJS checks all eight possible semantic transformations.
- If there is only one valid semantic transformation, it's applied and missed symbol becomes resolved.
- If there is more than one valid semantic transformations, the compiler reports an error instead.
- MetaJS can dive any depth of the entities graph and build chains like **request.session.user.username**.
- At the moment lookup depth is limited to 1 (each entitative transformation can use one entity only).
- Because Law of Demeter (LoD) or principle of least knowledge.

In the example bellow 2 arguments of **get-user-timeline** are missed. Symbolic transformation is used for resolving **limit**, entitative transformation with custom code generator – for **username**.

```
(entity request
  "Abstract web request with session."
  (has [session url])
  (rel [username] `(. ~sym 'session ~rel)))

(defn get-user-timeline (username limit)
  #"$username timeline: $limit")

(let* (req:request {session: {username: "me"}}
       limit 10)
  (get-user-timeline))
```

```
var getUserTimeline = (function(username, limit) {
  return ("" + username + " timeline: " +
          limit + "");
});

(function(req, limit) {
  return getUserTimeline(req.session.username,
                         limit);
})({session: {username: "me"}}, 10);
```

See more examples: https://github.com/dogada/metajs/blob/master/test/logos.mjs

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

5 Meaning of words

- Meaning of same name may vary in different source files.
- What does it mean **user**?
- Answer depends on context.
- Exactly as in real life, when word can have several meaning.
- The solution are namespaces: **db.user**, **twitter.user**, **app.user, etc.**
- Usage of names must be consistent only inside own namespace.
- When you import several namespaces that share same name, use aliases or fully qualified names.
- Exactly as with modules/packages of code.
- By default entities are defined in global namespace.
- And this is the only option now.
- Symbols and entities of your program in fact is a dictionary for the compiler.
- You teach compiler.
- Compiler understands you better.