MetaJS

Knowledge-oriented programming language

Dmytro Dogadailo
October, 2013
http://metajs.coect.net

Purpose

MetaJS is knowledge-oriented programming language. MetaJS compiler can generate source code, intentionally missed by programmer. This approach leads to loosely coupled and highly dynamic programs. MetaJS is a dialect of Lisp and provides many of the typical Lisp features including powerful macro system. It compiles to Javascript without runtime dependencies.

Function

Typical usage scenario: signature of a library function is changed but all arguments for new version of the function can be found in the call context. MetaJS can adapt all calls of such function to the new signature automatically. A programmer starts to use less nouns (local variables and function arguments) and more verbs (function calls). MetaJS compiler finds what does the programmer mean from semantic model of the program and surrounding context.

Motivation

Today compilers analyze only the grammar of the language and ignore the semantics of each specific program. As a result, we get the source code with high level of redundancy. With the increasing size of the program this naturally leads to difficulties in its support. Major changes in the application source code become virtually impossible without causing of new bugs and regressions.

Audience

The complexity of software and user's expectations increase much faster than programmer's productivity. MetaJS attempts to close this gap and offers to small teams and individual programmers a tool that allows to do more for less time. Semantic-aware compiler orthogonally combines with traditional Lisp meta-programming and provides new experience for all programmers who value own time.

Methodology

In addition to traditional macros that transform program at read- and compile-time MetaJS uses semantic transformations that runs after read- and compile-macros. In MetaJS functions can have required and optional parameters. When MetaJS compiler finds a function call without a required parameter it attempts to “resolve” it to make the valid function call.

The compiler tries to find all symbols and compound forms that can be used instead of missed function argument. The compiler replaces missed function argument only when it finds exactly one valid substitution. When valid substitution is not found or found more than one, compiler reports an error that programmer need to resolve manually. 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.

MetaJS uses two classes of semantic transformation: symbolic and entitative. During symbolic transformations compiler uses symbols available in the lexical scope of the function call instead of missed argument of the function. To qualify a symbol for the substitution of the missed function argument MetaJS tries to match name and meta information of the missed argument and each symbol available in the lexical scope of incomplete function call. Name of the argument can be matched with meta of context symbol and vise versa forming four possible types of combinations.

Let's see an example of symbolic transformation when name of missed function argument is matched with meta type of context symbol. On the left is MetaJS code and on the right is JavaScript code generated from MetaJS code. Result of semantic transformation is highlighted with **red** color. 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* (u:username "dogada")
      (get-user-timeline))
```

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

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

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**. The code **u:username** defines symbol **u** and declares that it is related to something called **username**. Meta types are not need to be defined at all. Compiler uses meta types and entities to build semantic model of the program.

Entities are defined with **(entity)** macro and used solely by compiler to understand relations between symbols. Entities are not types in statically-typed languages sense and have no corresponding JavaScript code. Entities are main building block of semantic model of the program.

Let's see an example of an entitative transformation. Such transformations always use an explicitly defined entity to find a valid relation between symbols available in the lexical scope of function call and missed function argument. Entities may define simple relation like property relation, for example **(has username).** Custom relation able to emit any form instead of missed argument, for example **(rel username `(. ~sym 'session ~rel))**, where **sym** is actual symbol matched to the entity and **rel** is target itself.

```
(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"});
```

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 **request** and entity **request** declares relation between **request** and **username** entities and **user-id** argument declared with meta type **username**.

You can find examples of all 8 types of semantic transformations used by MetaJS in the http://metajs.coect.net/pdf/metajs_semantic_code_transformations.pdf.

Conclusion

MetaJS is implemented in MetaJS, so even if it's still in alpha stage it's quite feature complete already. Famous Lisp macro system allows to add more features without need to change language itself. Anyone can try it without leaving a browser on http://metajs.coect.net/ or install as NodeJs module.

It's easy to predict that in large programs symbols and entities with same name may have different meaning in different source files even inside single project. When you import external libraries, you may expect even more name conflicts. MetaJS plans to solve this problem with namespaces and using of fully qualified names of entities inside compiler. A programmer can import and alias entities from other modules exactly like symbols are imported between namespaces.

Next obvious area of improvement is meta types inheritance and various checks of code validity based on semantic model of program.

The final goal is to make a compiler that can learn programmer's style at least particularly. A programmer concentrates own efforts on domain models design and architecture of the program. A programmer tells to MetaJS compiler what to do and teach it to understand program's entities and relations between symbols. Computer in real-time mode analyzes semantic model of the program, compares it with the code that programmer types and provides immediate feedback rather than shows blinking cursor.