GSoC 2017 proposal, core.match compilation

Contact Info

- Name: Sifou Biri
- Location: Winnipeg, Canada (Central Time Zone)
- Email: sifeddinebiri@gmail.com
- IRC/Slack: @sifou
- Twitter: @bsifou_
- Github: bsifou

Introduction

Core.match is a library that adds pattern matching functionality to *Clojure* language. Currently we generate a decision tree that get transformed into nested series of pattern/occurrence tests of Clojure forms and to benefit from smaller generated code-size we employ backtracking technique by throwing the same exception so that we only have one leaf-node for each action corresponding to our pattern clause matrix. On JVM side this poses little problems, as it optimizes this process by not producing the stack-traces associated with our exception, but on JavaScript engines this is very costly and that's the problem we will try to solve. The work will be on adapting and refactoring the structure of our current code to operate on transparent representation to generate different code, as it currently depends on the `deftypes` of the different patterns we have. This will result in faster and compact code running on our targets. And that also includes adapting different parts of our column selection system that chooses a column based on it's pattern types & positions.

We will be leveraging *clojure.spec* to parse and check the syntax of our user input patterns from the macro, this gives us a data representation of

the rows of patterns using regular but powerful Clojure structures. We will also get more flexibility at manipulating and shaping our information at each step of the process. Our work will also be consisting of 3 composable steps (producing matrix by transforming data user from macro and then passing it to our algorithm which will be producing DAG representation of Clojure code that will get used to generate Clojure code.

Some details

- Will be nice to expose the related information to our occurrences (like binding) and to also independently process some types of patterns that we have (like `or`, `app`) that is, we simplify our matrix before we start applying the algorithm - maybe with each pattern associate an occurrence (making it as the step after spec'ing the match and preserving it throughout the next transformations.)

- Remove the redundancy, there are many functions that we call many times specifically `regroup-keywords` (leaf-bind-expr, storing it on meta and as separate dispatch function) or in `cases` where we only need one matrix as input.

- Introduce reader-macros for clj/cljs specific code.

- Using one main representation and transforming it, making use of the help of multimethods dispatching on information associated with pattern (type) stored as part of data, currently we use protocols associated with `deftype`.

- Dispatch on matrix specialization, on most times, we are only specializing one row at a time. I think we can work on the assumption that we are working with a single row when specializing instead of a whole matrix (multiple rows) and handle matrix expansion separately from compilation step.

- I think at first I will try to do without the concept of grouping as sometimes it doesn't hold. When for example we have two consecutive rows with first patterns as map type with same `:only` keys that get expanded into a guard test with same semantic but different generated symbols in the test. So, If we handle rows of matrix separately and then specialize, that is expanding it into a form of simpler pattern

and each time providing a way to test the next level until we hit leaf-node by either succeeding the last test, or fail node (in current case we backtrack) which we will be doing it by jumping to the next main test as we will have multiple tests for first selected column patterns, with addition to a default one (added before to the matrix rep before feeding to decision tree producing function.) That leaves only one fail-node to be shared with all nodes (currently when we reach it, we backtrack to other ones on upper level.) Naturally we will have different levels, the main level ones where we have the default tests and the nested ones, although they will all be in one flat jump table.

Deliverables

- New version of core.mach that compiles to jump table and uses *clojure.spec* and transparent data structures.
- Documentation about how things fit together.
- Additionally if possible, customize the errors from *clojure.spec* to be more concise.

Commitment/Other info

- I will only be registered in one online course at my University (May-June)
- Without any problems, I'm able to work on it at least 8 hours a day.
- All my knowledge about Lisp, Clojure and functional programming is self-acquired. Started my journey first with ELisp then Scheme/Racket about a year ago, before focusing exclusively on Clojure starting last December.
- Previous experience also include writing a React app for a course project, knowledge of Java and JavaScript.
- Why Clojure? Awesome community. Awesome language.

Timeline

Before April 20

- Experiment more with *clojure.spec* and how it will be applied to our pattern syntax and the resulting structure when conforming it.
- Think more about the implication of implementing a jump table and what possible ways of approaching it.
- Continue studying and making more use of the REPL to play and experiment to get better feel of our current code.

April 20, May 30 (before official coding)

- Focus on the parts that are still not clear in *Marangent* paper & how it relates to our current implementation.
- Clarify the points I have not yet fully understood with my mentor, David Nolen.
- Share results of some experimentation and thoughts.
- Define the set of pieces of our current code that we will be reusing and try to work the details of how we will be refactoring the others.
- Collect more notes/questions

May 30, Jun 14

- Work on specifying the specs for our matching macro (making sure it's complete & harmonic by supporting the different syntax types of our patterns (like `:as` or `:when`)

June 14, June 30 (Phase I)

- Add any transformations to the resulted data structure as seen fit for our later needs when compiling.
- Make sure we have covered all matching patterns syntax.

June 24, July 13

- Add new functions/multimethods that works on this data structure which will allow us to specialize, expand, produce Clojure forms that we can test.
- Fix bugs as they will arise during process.

July 13, July 28 (Phase II)

- Work on adapting our algorithm to our new rep (like scoring system) this includes any addition of necessary information the algorithm might need to work properly.
- Experimenting and finding the most suitable and simple rep of our decision tree to generate a jump table from it.

July 28, August 6

- Make our algorithm target that representational, by implementing and adapting current code, the functions for our 3 main cases that get called recursively and their helpers.

August 6, August 18

- Maybe we will have to add an other step of transformation to analyze the tree structure resulted and produce a more suitable/close representation to generate cases from it.
- Work on steps for generating the code from it, that will be our main switch case, coupled with loop.

August 18, August 29 (Final)

- Work on fixing bugs and running/writing tests and making sure it's semantically compatible in every aspect with our current version.
- Documenting the different pieces and how they interact, enhancing comments. If possible, parse spec errors and try to provide alternative error messages focusing just on what went wrong.