Lisp in Summer Projects Submission

Submission Date	2013-10-19 01:21:00
Full Name	Akshay Srinivasa
Country	USA
Project Name	Matlisp-tensor
Type of software	library
General category	library
LISP dialect	Commmon Lisp
GitHub URL	https://github.com/enupten/matlisp
Did you start this project?	No, I'm modifying or extending an existing project.
Which file or directory contains the majority of your work?	Nearly all of the code has been rewritten using abstractions which were absent in the previous work.
Briefly describe your modifications	- Added a new template-like macro system - Added "template" code for BLAS-like methods, to generate native Lisp methods. This helps avoid overheads, and generalizes methods to arbitrary rings. - Ported old methods into the new framework. - Working on adding inline macro-reader.
Project Description	I want to describe my project in this form.
Purpose	The numerical library world of Lisp is fragmented and fractious, in the current state. My work on Matlisp is aimed at building a fast, abstract numerical array library, with nice abstractions.
Function	My work on Matlisp adds a new a template-like macro-dispatcher on type symbols, which is then used to encode symbolic knowledge of various operations on general arrays.
Motivation	Numerical languages of today, do not allow expressivity

which is commonly associated with Lisp. The Lisp world doesn't have a nice
numerical library either. There are plenty of wrappers around BLAS/LAPACK, but these aren't really enough. I've been working on
making Matlisp into a full-blown numerical library.

Audience

Computer Scientists/Mathematicians.

Methodology

Genericity is not very well supported in CL. CLOS of course can do this, but the demands of the CL-spec, means that avoiding method dispatch is impossible. For numerical code, where cache coherency is crucial, that performance suffers. One could in theory use macros to do something very clever, but there aren't defacto standards for doing this yet. What I've written is a very messy skeleton version of such a system. Here's an example for how the system works.

Define a generic macro-dispatch, which orders macroexpansion functions according to #'subtypep on the symbol-type "ty".

```
(deft/generic (t/f+ #'subtypep) ty (&rest nums))
```

Define a generic method on numbers.

```
(deft/method t/f+ (ty number) (&rest nums)
(let* ((decl (zipsym nums))
(args (mapcar #'car decl)))
`(let (,@decl)
(declare (type ,ty ,@args))
(cl:+ ,@args))))
```

Define a generic method on strings.

```
(deft/method t/f+ (ty string) (&rest nums)
`(concatenate 'string ,@nums))
```

Once defined, these can be used by giving type arguments.

```
(t/f+ string "ad" "sadas") => "adsadas"
```

With careful use of these macros, it becomes easy to build up abstraction upon which to write very fast code. It also becomes
easier to work with arrays, for instance the specialized macro:

```
(einstein-sum real-tensor (i j k) (ref C i j) (* (ref A i k) (ref B k j))
```

----

generates the 3-loop naive lisp code for gemm, which runs at about ~1.5 times the speed of compiled C code (in SBCL). The hope is that
with more work, I will be able to build a DSL for specifying operations on arrays seemlessly, when vectorized operations in BLAS are not available.

Unlike other languages such as Numpy/MATLAB, Lisp enables the use of very handy macros for talking to libraries in Fortran/C, with the least amount of boiler-plate code. It also enables access to fast-callbacks, which are generated by the Lisp compiler. See
src/packages/odepack/dlsode.lisp for an example.

	Of course, the generic macro itself is very unsafe, and tends to be hard to debug, for the uninitiated; but once the required capabilities are known, it is easier to beautify the framework. The work is very far from completed, because the lack of easy accessibility to common functions, but hopefully this will change with time.
Conclusion	After many iterations, a template-like system was found to be a feasible abstraction for encoding symbolic-numerical information for writing automatically generated numerical library. The work is still far from being complete, but I think the ideas in the project are useful things in the long run (atleast for the author's reasearch :). The hope is that, it becomes much less painful for doing symbolic-numerical computing in the future.
Build Instructions	See the "How to Install" section in the README file.
Test Instructions	Because this is an as yet incomplete library, one could check out example code in src/tests/.
Execution Instructions	Here's an example: (let ((A (randn '(10 10))) (B (randn '(10 10)))) (gemm 1 A B 0 (zeros '(10 10))))
Describe any bugs or caveats	Some methods may have silly bugs. If you find any, please report them to the Author.
Official	I have read rules and have abided by them. I am 18 years of age or older. I am not living in Brazil, Quebec, Saudi Arabia, Cuba, Iran, Myanmar (Burma), North Korea, Sudan, or Syria.