Bringing Software Transactional Memory to Julia

18.337

David Girardo

daig@mit.edu

Motivation

- Parallel programming is held back programming complexity
- MIMD ergonomics lag behind SIMD
- Manual locking still the go-to solution
  - Not maintainable / Reusable
  - Painful!

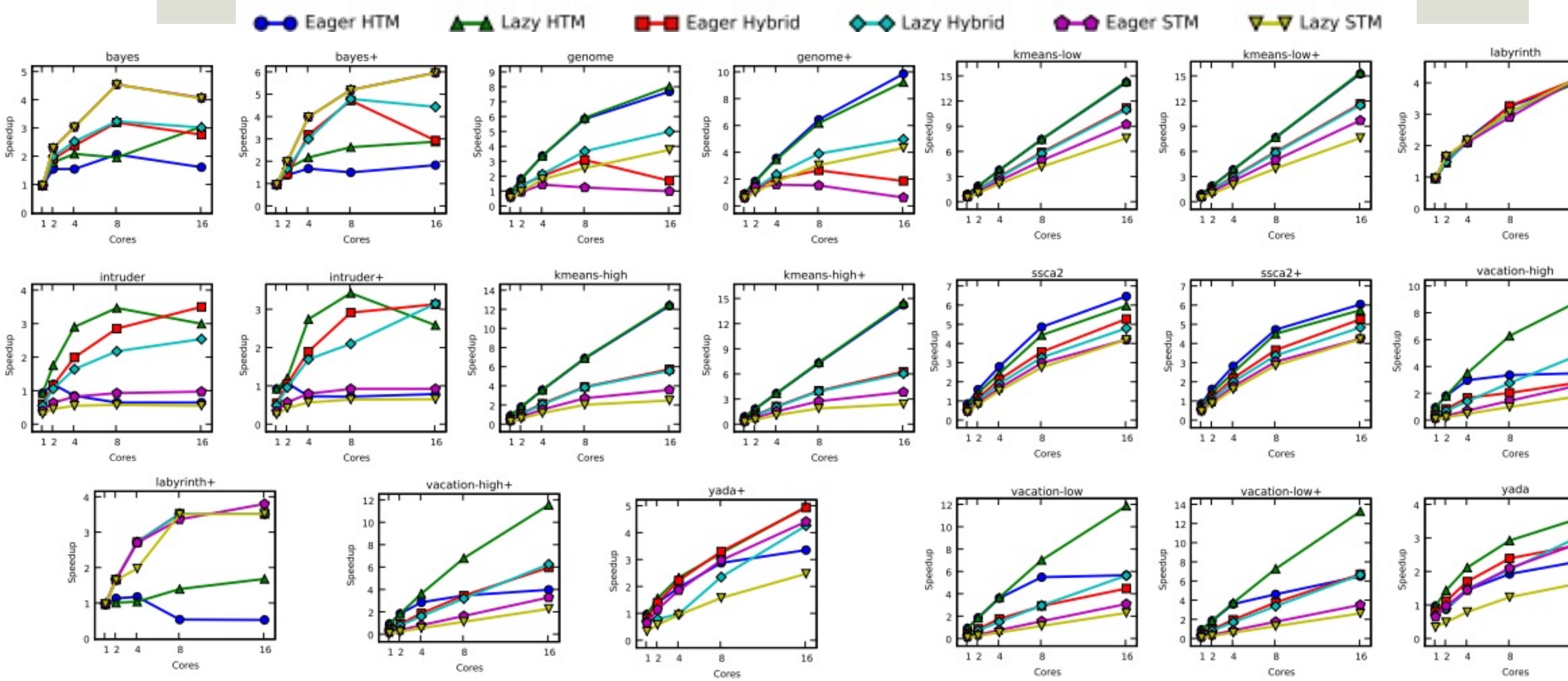
Software Transactional Memory

- Idea: Replace Locks with Atomic Blocks
  - To the shared environment: Every operation in a block occurs, or none do
  - To the local environment: ???
- TVar / Managed References
  - A memory cell that can enforce atomic transactions
  - "They're like regular references, except not broken" – Rich Hickey

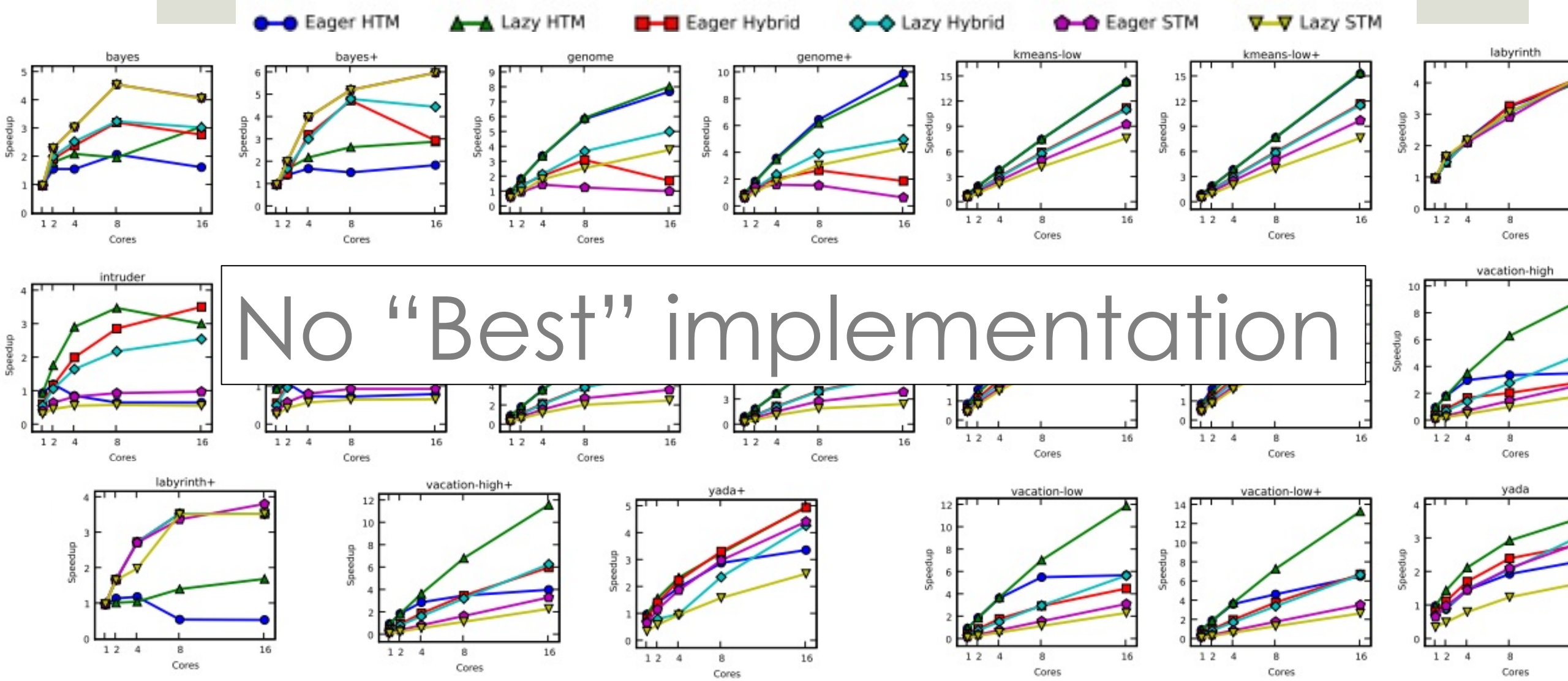
```
abstract Tvar{T}
function readTVar{T}()::T
function writeTVar{T}(v::T)
```

STM implementations

- Common idea:
  - Attach version number to TVar
  - If another process touched the variable before commit, abort
- Many choices for scheduling, logging, and heuristics,
  - Each with a tradeoff
  - Most implementations are monolithic



Stanford Transactional Applications for Multi-Processing (STAMP) benchmark
http://csl.stanford.edu/~christos/publications/2008.stamp.iiswc.pdf



No "Best" implementation

Stanford Transactional Applications for Multi-Processing (STAMP) benchmark
http://csl.stanford.edu/~christos/publications/2008.stamp.iiswc.pdf

Goal

- Framework for STM syntax
- Explore Julia's unique design space
  - Efficient Types
  - Macros
  - Leverage existing parallelism ecosystem

Transactional Variables

- Pros:
  - Simple
  - Transparent
- Cons:
  - Complicates algorithm
  - Solution: Macros

```
type TVar{T}
        v::T
        version::Int
        varID::Int
end
AtomicBlock = Dict{Int,Int}
function beginAtomic()::AtomicBlock
function commitAtomic(d::AtomicBlock)
```

Sweet Macro Sugar

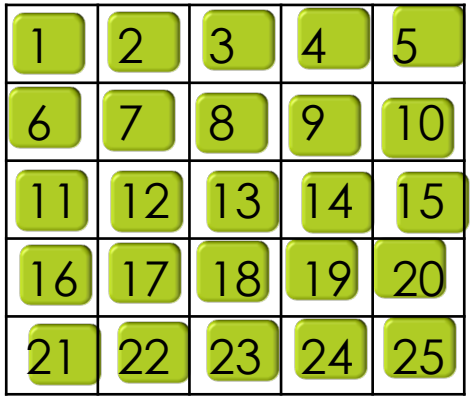
- @dosync <your code>
  - Convert all assignments to .writeTVar, reads to .readTVar
- Pros:
  - Trivial code modifications
- Cons:
  - Many options to pick (boundaries, backends), hard to “do the right thing”
- Fixes:
  - ‘@atomicVar x = 3’: Define Tvar, mark which names to convert in @dosync
  - Backend agnostic intermediate

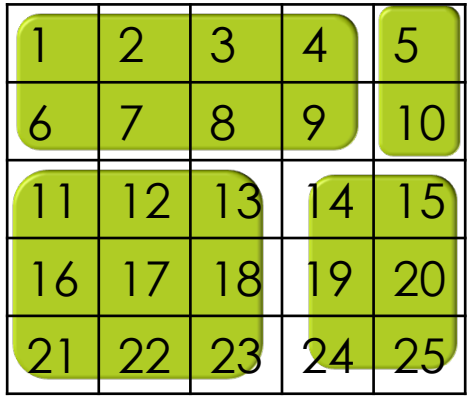
Task STM

- Tasks
  - AKA “Green Threads”
  - AKA Coroutines
- Pros
  - Light, Fast
  - True Shared Memory
- Cons:
  - Concurrent but not Parallel

Shared Array STM

- Shared Arrays
  - A thin wrapper around Julia's Shared Arrays, array "chunks" tracked by TVars
  - Discrete chunks (track every index, lots of overhead)
  - Indiscrete chunks (track whole array, misses parallelism)
- Pros:
  - Easy to work with
  - Efficient Sharing
  - Scalable
- Cons:
  - Only works for arrays (for now)
  - Determining data dependence ("chunking") is hard

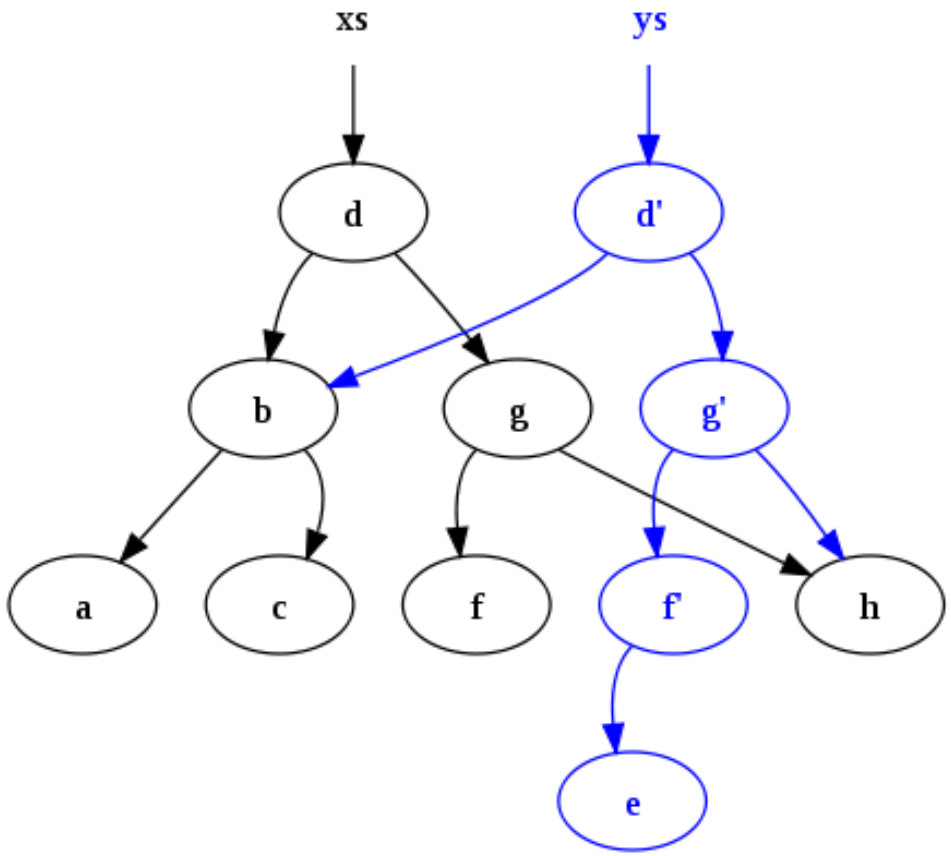




??

Optimizing with Persistent Data Structures

- Preserve old versions of data
- Minimize data duplication
- Challenges
  - Normally either baked into language, or manual and tedious
  - Works for boxed data
    - Unboxed Speed possible with clever records



“STM as a library”

- STM usually requires a single, fixed implementation language support
  - Haskell, Clojure:
    - persistent data-structures
    - values boxed by default
    - GC already tuned
  - C++, Java family use special-purpose compiler
- While Julia has:
  - Types that are no different from primitives
  - Fully Expressive macros
  - We should have the freedom to choose a backend!

Future Work

- In-the-pipeline
  - Distributed backend
  - Backend-Agnostic intermediate, ClusterManager style
  - Backend benchmarking
- Julia Community:
  - Reduce overhead with “Chunked” shared arrays
  - Proper interfaces / function types would be *really* nice
- Research-Grade:
  - Backend Heuristics
    - Julia-level static program/architecture analysis (FFTW, PetaBricks)
    - JIT in LLVM
  - Transform programs to use Persistent Data Structures

Resources \\ Questions

- http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey
- http://blog.enfranchisedmind.com/2009/01/the-problem-with-stm-your-languages-still-suck/
- http://chimera.labs.oreilly.com/books/1230000000929/ch10.html