Implementing a Vau-based Language With Multiple Evaluation Strategies

Logan Kearsley
Brigham Young University

Introduction

Vau expressions can be simply defined as functions which do not evaluate their argument expressions when called, but instead provide access to a reification of the calling environment to explicitly evaluate arguments later, and are a form of statically-scoped fexpr (Shutt, 2010). Control over when and if arguments are evaluated allows vau expressions to be used to simulate any evaluation strategy. Additionally, the ability to choose not to evaluate certain arguments to inspect the syntactic structure of unevaluated arguments allows vau expressions to simulate macros at run-time and to replace many syntactic constructs that otherwise have to be built-in to a language implementation.

In principle, this allows for significant simplification of the interpreter for vau expressions; compared to McCarthy's original LISP definition (McCarthy, 1960), vau's `eval` function removes all references to built-in functions or syntactic forms, and alters function calls to skip argument evaluation and add an implicit `env` parameter (in real implementations, the name of the environment parameter is customizable in a function definition, rather than being a keyword built-in to the language).

McCarthy's Eval Function
(transformed from the original M-expressions into more familiar s-expressions)

```
(define (eval e a)
  (cond
   ((atom e) (assoc e a))
   ((atom (car e))
    (cond
     ((eq (car e) 'quote) (cadr e))
     ((eq (car e) 'atom)  (atom   (eval (cadr e) a)))
     ((eq (car e) 'eq)    (eq     (eval (cadr e) a)
                                  (eval (caddr e) a)))
     ((eq (car e) 'car)   (car    (eval (cadr e) a)))
     ((eq (car e) 'cdr)   (cdr    (eval (cadr e) a)))
     ((eq (car e) 'cons)  (cons   (eval (cadr e) a)
                                  (eval (caddr e) a)))
     ((eq (car e) 'cond)  (evcon. (cdr e) a))
     ('t (eval (cons (assoc (car e) a)
                     (cdr e))
               a))))
   ((eq (caar e) 'label)
    (eval. (cons (caddar e) (cdr e))
           (cons (list (cadar e) (car e)) a)))
   ((eq (caar e) 'lambda)
    (eval (caddar e)
          (append (pair (cadar e) (evlis (cdr e) a))
                  a)))))
```

Vau Eval Function

```
(define (eval e a)
  (cond
   ((atom e) (assoc e a))
   ((atom (car e))
    (eval (cons (assoc (car e) a)
                (cdr e))
          a))
   ((eq (caar e) 'vau)
    (eval (caddar e)
          (cons (cons (cadar e) 'env)
                (append (pair (cadar e) (cdr e))
                        a))))))
```

Of course, to be useful, the environment must be pre-filled with implementations of basic built-in functions and syntactic forms, so the complexity of the complete language interpretation does not disappear- it is simply moved out of the main evaluation loop and into the standard library. The number of basic functions that must be provided is, however, surprisingly small; even `quote` can be implemented entirely in-language as a very simple vau expression which simply returns its unevaluated argument, ignoring the reified dynamic environment: `(vau (e) env e)`.

This same kind of interpreter transformation can be done given first-class macros as a basic language feature instead of vau expressions[1]; however, this requires lambdas and macros to both be provided independently by the language implementation. Vau-based languages are more flexible because they can implement lambdas and macro systems as syntactic sugar. Unlike other implementations of fexprs, vau

1 Matt Might demonstrates just such a transformation in a blog post on implementing first-class macros in a metacircular evaluator: http://matt.might.net/articles/metacircular-evaluation-and-first-class-run-time-macros/

expressions are therefore not implemented as a separate special type co-existent with lambda functions, macros and quotation operators; rather, vau expressions are the material from which all other evaluation-controlling structures are built.

The behavior of macros can be simulated by wrapping a body expression that performs some manipulation on the syntactic structure of a vau expressions arguments in a single call to `eval` the result in the context of the reified dynamic environment. It is fairly trivial to write a macro-generating vau expression that uses this technique to create other macro-like vau expressions given the syntax for a macro body to insert into the wrapper of a vau that evaluates the result of its body, as demonstrated in Appendix B.

Lambdas are produced by means of a `wrap` construct, which takes a vau expression as input and produces a new function, called an applicative, that guarantees evaluation of argument expressions before passing the argument values to the underlying vau expression. Wrapping can be applied multiple times to result in multi-fold applications of `eval` to argument expressions.

The `wrap` construct can be implemented entirely within the language by simply mapping `eval` over the list of arguments; however, this approach eliminates the possibility of recovering the underlying vau expression from an applicative; to allow for deconstruction of applicatives, `wrap` is introduced as a built-in applicative constructor, with a corresponding accessor function `unwrap` that removes an evaluation step from an applicative.

When `wrap` is introduced, the language gains a built-in evaluation strategy. Using `wrap` and vau together to produce lambdas thus involves a tradeoff between familiarity and code conciseness on the one hand, and control over evaluation strategy on the other. Vau languages can provide multiple implementations of `wrap` (embodying multiple different evaluations strategies) to allow programmers to manage this tradeoff. In this project, I have implemented a language called vernel (a play on John Shutt's Kernel, 2009) which provides left-to-right, R6RS-compliant (Sperber et al., 2010), fork/join, lazy, and future-based versions of `wrap`.

Semantics

I designed vernel's semantics in PLT Redex[2]. While the machine implementation of the vau evaluator is fairly simple, a syntax-based specification of semantics is not. This is largely due to the unanalyzability of vau: since a vau expression can choose arbitrarily to evaluate arguments or not, it is impossible to distinguish code from data without actually running the program and observing its dynamic behavior. As a result, variable references, call sites, and other types of abstract syntax nodes whose behavior must be specified in the semantics cannot be statically identified. One solution is to reduce input programs, which can consist solely of lists and symbols, with no other kinds of syntax nodes, in multiple steps. First, the outermost list or symbol is transformed into the appropriate more-specific abstract syntax node; reduction then proceeds according to the semantics of that node type.

Most of these intermediate node types cannot be entered by a programmer (i.e., have no literal form, but must be generated during the execution of the program); these are hereafter referred to as “internal syntax”, contrasted with “external syntax” which can be written down in source code. Some internal syntax is present in most Lisp implementations, because there is no string of characters a programmer can write down that will cause `read` to produce, e.g., a procedure[3]; but, while it is rare in the actual vernel implementation (being limited to procedures), it is pervasive in vernel's formal semantics.

The impossibility of statically identifying variable references also means that a standard ahead-of-time substitution model with alpha-variance for evaluating function applications cannot work: no symbol within a function body can be guaranteed to represent a variable reference, and thus be replaceable, until it is actually evaluated. This, combined with the fact that programs must have access to explicit reified environments, means that the syntactic specification of semantics must itself include an explicit representation of the current environment for any expression. Evaluation can only occur in the presence of an environment for resolving references, and so programs are defined as a pair of an evaluation-triggering environment with an expression. Expressions are defined as any syntax node other than a program, and data or values are defined as expressions that contain only external syntax (i.e., lists and symbols, or other built-in atomic data types like booleans and

2 As the complete semantics is too large to lay out here, see Appendix A for a link to the PLT Redex code.
3 Clojure is a notable exception to this rule. See http://clojure.org/reader

numbers).

Lists paired with environments are reduced by generating a call site node with the rule

C[(Env (list Value …)))] → C[(Env (call Value …)))]

and recursively reducing the list head. If reducing the list head puts a `wrap` node of any kind in a call site position, individually-specified reduction rules for each `wrap` type determine the evaluation strategy. In general, argument evaluation is accomplished by copying the current environment and transforming every argument expression into a pair of environment and expression, producing new reducible sub-programs and transforming the surrounding context into a non-reducible non-expression until all sub-programs are eliminated.

Interestingly, Felleisen, Findler, and Flatt (2009) go to great lengths to specify R6RS-compliant argument evaluation behavior without any recourse to internal syntax. The argument against using internal syntax in this case is weak given that internal syntax is already widely required to describe the most basic functionality of the language, and the description of R6RS wrap using wrap-specific internal syntax is surprisingly straightforward. This is done by replacing argument expressions with thunks, and then transforming thunks back into sub-programs one at a time.

The existence of thunk nodes containing an environment and an expression also provides a first step towards implementing lazy wrap. Thunks can simply be added to the environment and transformed into a subprogram when accessed. Unfortunately, because environments must be copied, this approach may result in incorrectly evaluating a single original thunked expression multiple times in different places. Specifying the proper semantics for laziness requires the addition of the notion of a heap and pointers to the semantics. Whenever a thunk must be generated, the call site is wrapped in a new heap which contains the thunk matched to a globally-unique pointer value; this pointer value is then added to the environment. When a pointer is accessed, the associated thunk is transformed into a subprogram for evaluation. When a heap pointer is paired with a value, the guarantee of global uniqueness means that the resulting value can be immediately substituted for all occurrences of that pointer in the whole program. Heap-rewriting rules lift newly generated heaps to the highest syntactic positions possible and merge adjacent heaps together to maintain composability.

Futures make use of the same heap-indirection mechanism as laziness, but insert subprograms directly into the heap for immediate evaluation rather than delaying them with thunks. This is directly analogous to the machine-based semantics for futures described by Flanagan & Felleisen (1994).

Continuations

The language as implemented contains first-class continuations. These actually have a significant impact on the structure of much of the interpreter, and mainly the implementations of wraps; different kinds of wraps require a profusion of different kinds of continuations. However, due to their complexity, I have chosen not to model continuations in the formal semantics. As described in the Implementation section, each version of `wrap` (with the exception of left-to-right) requires two different kinds of continuations; additionally, a context-rewriting metafunction would be required to replace initial continuation with subsequent-activation continuations whenever a continuation is encountered either implicitly or explicitly in the reduction process. These additions would greatly increase the size of the semantics specification, significantly reducing comprehensibility with little pedagogical value.

Side Effects and Mutation

The effect of different evaluation strategies can only be observed in the presence of side-effects or non-termination. Continuations and mutation, both of which are available in the language implementation, are the most obvious potential sources of side effects. However, neither of these are available in the formal semantics. Thus, in order to provide test cases that can demonstrate different semantics for different evaluation strategies, an error form was added to the semantics and the language implementation that aborts evaluation when it is encountered.

Mutation is available to the language implementation because environments are not actually copied-only pointers to them are. Implementing mutation in the formal semantics would require using heap indirection

on environments. The most straightforward way to do this would be to move all values into the heap and store only pointers in environments, while also eliminating the replacement rules for heap entries that resolve to values, instead replacing pointers with values only when they are actually accessed. This corresponds to value boxing in machine implementations. Variable lookups would then become a two-step process of replacing variable references with pointers followed by pointer accesses, and constructing a new environment would be complicated by the need of simultaneously generating new pointers and heap entries.

Implementation

Go was chosen as the initial implementation language for the vernel interpreter due to its built-in concurrency support. In hindsight, meta-concurrency in Go was not as useful for implementing concurrency in vernel as expected, largely because of the Go runtime's automatic deadlock detection; when it is determined that any concurrent thread can never proceed (e.g., as when an argument expression to a concurrent applicative activates a continuation to a higher scope, thus ensuring that argument evaluation never completes and the function body cannot run), the Go runtime panics and terminates the entire program rather than silently garbage collecting the dead thread. Blocking and unblocking of thread execution when waiting on data dependencies therefore had to be implemented from scratch.

The core of the vernel interpreter is a simple loop which acts on three registers containing the current expression, environment, and continuation. While Go implementations are permitted to do Tail Call Elimination, it is not required; in practice, this means that the Go compiler doesn't. Implementing tail call semantics for vernel therefore required trampolining. Internally, function calls and continuation activations take a pointer to a structure containing the vernel machine registers, modify the registers, and then return a boolean flag to the `eval` loop to indicate whether the current expression should be evaluated, or simply passed directly as a value to the current continuation. This trampolining means that `eval` is only called once at the beginning of each concurrent thread.

Ensuring that library functions that start new threads (mainly `wrap` implementations) have access to `eval` for that purpose required some slightly unwieldy architectural decisions to work around Go's module system and compilation model. The `eval` function needs access to vernel type definitions, but type implementations (such as the Future object) also need access to `eval`, and circular dependencies are prohibited in Go. To work around this, all interpreted function calls take an extra argument which is a reference to `eval`, injecting the dependency at run-time to work around the module system restrictions.

Several different kinds of internal data types are exposed to the interpreted language as callable functions, including built-in native functions, in-language vau expressions and applicatives, continuations, booleans (which behave like Church-encoded booleans, eliminating the need for explicit conditionals) and environments (obviating the need for a separate in-language `eval` function by treating environments as functions that evaluate expressions in the context of themselves). Rather than requiring a type-switch case for each of these different data types, the core interpreter is kept simple by taking advantage of Go's interface-based polymorphism, and simply defining an appropriate `Call` method on each type. Built-in callables, however, all behave as raw vau expressions; for maximum flexibility and to avoid re-implementing argument-evaluation trampoline calls in almost every case, reified environments and continuations are provided to the interpreted language (by the function call mechanism for reifying environments and the implementation of the `bind/cc` built-in function for reifying continuations) as pre-wrapped applicatives, catering to the most common use case, which may be unwrapped if desired; e.g., to implement namespacing by passing raw source code to an unwrapped environment, or to pass a quoted value to an continuation.

Implementing Wraps

The implementation of left-to-right wrap is unique in that it connects argument continuations together in sequence, thus ensuring that activating an argument continuation a second time will re-evaluate all argument expressions to the right, rather than simply ensuring that they are evaluated exactly once in the proper sequence. This is necessary to implement `begin` as a `wrap` of `list`. All other wraps are conceptually concurrent and have continuation semantics such that secondary activations of any argument continuation will result in immediately re-evaluating the function body with all other argument positions filled with their original values.

This requires that argument continuations behave differently between the first and all subsequent activations. The behavior of arguments in each kind of wrap is subtly different.

For *fork/join wrap,* the first activation of an argument continuation must fill in the value for its slot, then check if all other slots are filled. If they are, any blocked threads are re-activated, and the current thread is re-used to continue evaluation of the function body. This may result in the original thread used to make a function call dying and being replaced by a completely new thread depending on the order in which argument expressions finish running. If other slots are not filled, the thread simply dies. On subsequent activations, the current thread must either copy the saved original argument values, replacing the value in its own slot, and then re-evaluate the function body, or, if some slots are unfilled, block.

The fact that executing a fork/join wrapped function may change which thread is in charge of evaluating the remainder of the program means that it is useless for the `eval` function in Go to return a value to its caller; the final result returned to any particular instance of the `eval` loop may turn out to be the result of any arbitrary argument evaluation anywhere in the subprogram it was originally given to evaluate. Communication with the top-level REPL is instead accomplished by passing a special initial continuation to `eval` which captures a channel over which the evaluation result may be sent to the REPL thread.

For *R6RS wrap*, expressions are guaranteed to complete in some linear order; no real concurrency occurs. Therefore, initial continuation activations simply result in the argument thread dying, and all subsequent evaluations can copy existing argument values and proceed to re-evaluate the function body.

For *lazy wrap* and *future wrap*, argument slots are filled in with placeholder deferred-value objects and function body evaluation begins immediately. Initial continuation activations must resolve the deferred value, while subsequent activations must replace the appropriate argument slot with a real value and re-evaluate the function body from there. Like fork/join, using futures may result in swapping main program execution between different threads as execution may block waiting for a future value (resulting in the original thread being garbage collected) and then resume on a different thread when the future value is resolved.

These behaviors are implemented internally with self-mutating continuations, with the initial continuations overwriting function pointers to themselves with pointers to the appropriate subsequent implementations when they are activated. Because of the differences in each version of wrap, there is a great profusion of different kinds of continuations. Go's support for closures with mutable captured variables made it very easy to construct any necessary continuation on-the-fly; however, the non-inspectability of Go closures resulted in some additional complications, such as the need for continuations to properly replace themselves in the vernel machine registers rather than having a generic push/pop stack interface available, and the eventual need to store extra copies of all captured variables on the side so that they could be accessed by the memory profiler. Ideally, closure lifting would be used to implement each polymorphic continuation variant as an explicit data structure.

Additionally, vernel would ideally make the use of laziness and futures entirely transparent, handling deferred-value objects entirely in the background such that the programmer only ever sees normal values. This of course would require inserting strictness points into the implementations of all built-in atomic functions. For ease of testing and development, the language currently requires explicit calls to `strict` to be inserted by the programmer at all strictness points except function call positions and argument list positions (distinct from argument positions- if a deferred-value evaluating to a list is the `cdr` of a list interpreted as a function call, it will be forced; laziness in individual argument expressions will be propagated). This can result in the semantics of a function changing if it is unwrapped and re-wrapped with laziness or futures.

Compilation Strategies

Given that vau cannot be statically analyzed, meaningful compilation may initially seem impossible. Recall, however, that vau expressions can trivially implement run-time macros; indeed, evaluating a vau expression that does not evaluate any of its arguments is equivalent to macro expansion (for certain kinds of macros) (reference to Shutt thesis), which is typically a compile-time activity. Macro expansion can be generalized to partial evaluation (ref “A Hacker's Introduction to Partial Evaluation”) (used in nearly all modern compilers as an optimization technique in the form of function inlining and constant folding) as a potential route to meaningful compilation of vau expressions.

Because of the high degree of internal syntax in vernel's semantics, a partial evaluator is essentially a source-to-source transpiler to a new language that promotes the majority of vernel's internal syntax to external status and contains the vernel language itself as a subset, sectioned off by calls to `eval` (environment applications). This new language could then be run directly in an extended, more efficient interpreter. However, the statically-analyzable subset of the new language would itself be amenable to many more traditional compilation techniques for functional languages, such as those described by Dybvig (ref "implementation strategies for scheme"). The modularity available to the vernel `eval` function extends to the implementation of such a transpiler, which can in fact use the standard `eval` loop given compile-time implementations of standard library functions which know how to deal with missing input values.

Intuitively, it seems that the majority of real vau programs would be reducible to fully-analyzable residuals[4] with no remaining applications of dynamic environments, resulting in the possibility of producing native binary executables that are just as efficient as those produced by any other compiled Lisp. Further research needs to be done to validate this assumption. When dynamic evaluation is present in the residual program, it is necessary to ensure that the interpreter and any necessary library functions are still available at run time. If the compilation target is a vernel-specific virtual machine, it can designed such that the `eval` loop and standard library will always be available by default. In other cases, however, it would be necessary to link the interpreter and standard library into the final executable. This could result in a lot of bloat and the inclusion of dead code since it cannot be determined what library functions may be accessed during dynamic evaluation. But, while it is impossible to determine ahead of time which symbols will eventually be evaluated as variable references, it is known that all possible variable references will derive from symbols present in the source code. Thus, a conservative optimization can be done by scanning the residual program for all its symbols and trimming any symbols not present in that set from all reified environments (including the standard library) prior to linking.

When vernel is interpreted, significant performance improvements can often be gained by providing native standard library implementations of functions that could also be trivially implemented in the language. After compilation by partial evaluation, this is less true, but the compilation process itself can be made more efficient and allow for more informative error messages by providing native compile-time implementations of some constructs; e.g., a native implementation of `quote` can simply directly return its argument, bypassing the usual function call machinery for reifying environments and binding arguments to formal parameters.

Performance

A standard test suite of basic functionality (to ensure correctness) and the computationally expensive operation of computing a list of square roots was run with basic functions using one of four different kinds of wrap to investigate their comparative performance, and thus provide some idea of whether having the variety available is actually practically useful. (The R6RS wrap implementation was not profiled because it is identical to left-to-right wrap in requiring strictly sequential argument evaluation despite being conceptually concurrent.) The test suite initiates the square-root computations before running correctness tests, and then requires the values at the end, giving the maximum opportunity for parallel computation during the correctness tests. Intuitively, one would expect that more highly parallel executions would require less total run time and greater amounts of memory to accommodate the memory requirements of more than one calculation at the same time, but with sub-linear scaling due to memory sharing between threads.

Profiler

The Go profiling tools turned out to be not very useful for measuring the performance of the interpreted code, due to a combination of lack of awareness of interpreted functions (most computation in the Go code happening in a small number of interpreter functions, such as the main `eval` loop) and unpredictable garbage collection introducing noise into estimates of actual memory requirements at any point. Thus, the interpreter was instrumented with its own profiler, and functions were added to the standard library to activate and

4 As all of the profiled test cases require no external input and do in fact produce output values, the performance experiments presented in this paper may reasonably be construed as measuring the performance of a theoretical vernel compiler which outputs fully-analyzable single-value programs.

deactivate profiling of interpreted code so as to avoid recording profiling data for the initialization stage (defining all of the necessary in-language library functions required to run the actual tests).

The profiler collects data on how the interpreted code would have run on a machine with an infinite number of parallel processors, and with a simulated memory model that provides one slot for every language-primitive type, and one slot for every primitive contained within another complex primitive type (such as a closure, environment, continuation, or list). Each thread keeps track of how many cycles it has made through the `eval` loop, and this is the basic unit of simulated time and work; when new threads are created, their counters are initialized to the clock value of the parent thread. The total simulated run time is the maximum number of cycles recorded by any thread, while the total work done is the total number of cycles executed in all concurrent threads. Deadlocked threads are assumed to be eliminated by a sufficiently optimizing compiler, and thus do not hold memory as far as the profiler is concerned after they block. Blocked threads, however, have their clocks fast-forwarded when they become unblocked, and contribute to the recording memory requirements during waiting periods.

Results

	Left-to-right	Fork/join	Lazy	Futures
Total Cycles	4179	1995	4991	2897
Total Work	4530	4434	5680	5738
Normal Cycles [5]	2.095	1	2.502	1.452
Normal Work	1.022	1	1.281	1.294
Max. Memory	2938	5805	3752	6499
Max. Threads	2	9	5	10
Avrg. Threads	1.084	2.223	1.138	1.981

More detailed graphs of parallelism and memory consumption over time for each test run are listed in Appendix D.

The basic expectations are born out- programs with higher degrees of parallelism do exhibit higher peak memory and generally faster run times. Futures can take advantage of slightly more parallelism than fork/join argument evaluation, but not by a large margin; this can probably be attributed to the relatively small bodies of functions used in the test suite which allow for minimal temporal overlap between evaluation of argument and body expressions. Examining the charts in Appendix D further reveals that the test run using futures not only spends very little time running 10 threads in parallel, but actually spend less time at the 9-thread parallelism level than the run with fork/join. Because function-body parallelism could not be extensively exploited, the overhead of handling future objects overwhelms the potential parallelism improvements to produce significantly longer run-times than what you can get with just fork/join.

Because all values are eventually forced at the end of the test suite, we also don't see much advantage to lazy evaluation. Instead, we simply have to deal with the overhead of creating and forcing thunk objects with no reduction in the useful computation performed. This makes laziness look like the worst possible choice for this test suite, as a simple left-to-right evaluator beats it in speed, work, and memory. It is of course simple to find real programs in which laziness is on the whole beneficial, but this test suite starkly demonstrates the overhead that must be overcome.

One might reasonably expect that the total work performed by programs using left-to-right vs. fork/join wraps should be the same, as they do not produce any extra house-keeping objects or require extra steps for forcing or unboxing values. However, fork/join ends up doing slightly less work overall because of the differing continuations that must be produced. When a fork/join argument continuation is activated, it kills the current thread without returning control to the `eval` loop, unless it is the last-completing thread chosen to continue

5 Normal values are total values divided by the minimum total value for that measurement and show overhead accumulated above the minimum baseline performance.

with function body evaluation. When a left-to-right argument continuation is activated, it always returns control to the `eval` loop, thus incurring an extra virtual clock cycle.

Futures and laziness might also be expected to have identical overhead for creating and late forcing deferred-value objects. Again, the slight discrepancy is explained by how often certain operations return control to the `eval` loop. The continuation chain for evaluation of a lazy thunk is spliced directly into the continuation chain for the thread that initially forced the value, and thus has the same performance characteristics as inserting the deferred expression directly into the source code at the strictness point, plus one extra cycle for the call to `strict`. Resolving a future value, however, may require additional cycles due to the potential need to block execution and then resume on a different thread after waiting for future evaluation to complete.

Using vau and wrap to provide flexibility in the choice of evaluation strategies eliminates one additional potential source of parallelism: because determining the evaluation strategy to use on argument expressions depends on the type of the procedure being applied, the procedure expression must be evaluated strictly before any arguments. In practice, however, this is not a serious loss, as the vast majority of procedure expressions are simple variable references.

Because it is rare to find a consumer-grade computer with 9 or 10 cores available for parallel user-space execution, the simulated execution speeds seen for fork/join and futures are unlikely to be encountered in real near-term applications. When the computer architecture does not allow for full parallelism, there is some overhead to creating additional threads; however, non-preemptive scheduling and the use of a thread pool can minimize that cost, and these results demonstrate that even relatively small programs can contain sufficient inherent parallelism to take advantage of fairly large CPU resources when they are available.

References

Dybvig, R. K. (1987). *Three implementation models for scheme.* (Doctoral dissertation). Retrieved from http://www.cs.indiana.edu/~dyb/pubs.html

Felleisen, M., Findler, B. F., & Flatt, M. (2009). Case study 1: Order of evaluation. In *Semantics engineering with PLT redex* (pp. 259-270).Cambridge: The MIT Press

Flanagan, C., & Felleisen, M. (1994). The semantics of future. Rice University Comp. Sci. TR94-238

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM, 3*(4), 184-195. Retrieved from http://www-formal.stanford.edu/jmc/

Sperber M., Dybvig R. K., Flatt M., van Straaten A., Findler R. and Matthews J. (Eds.). (2010) *Revised^6 report on the algorithmic language scheme*. Cambridge: Cambridge University Press.

Shutt, J. N. (2009). *Revised -1 report on the kernel programming language.* Unpublished manuscript.

Shutt, J. N. (2010). *Fexprs as the basis of lisp function application or $vau : The ultimate abstraction.* (Doctoral dissertation). Retrieved from http://www.wpi.edu/Pubs/ETD/Available/etd-090110-124904/

Appendix A: Source Code

For the PLT Redex semantics, see
https://github.com/gliese1337/CS598R/blob/master/semantics/redex-semantics.rkt
For the interpreter source, see
https://github.com/gliese1337/CS598R/tree/master/src

Appendix B: Sample Code

The implementation of `let`

```
(def let (vau (bindings body) eval
    (eval (cons (list vau (map car bindings) (q #ignore) body)
                (map cdr bindings)))))
```

The implementation of `cond`

```
(def cond (vau opts eval
   ((nil? opts) ()
          ((eval (caar opts)) ;(must evaluate to a boolean)
           (eval (cadar opts))
           (eval (cons cond (cdr opts)))))))
```

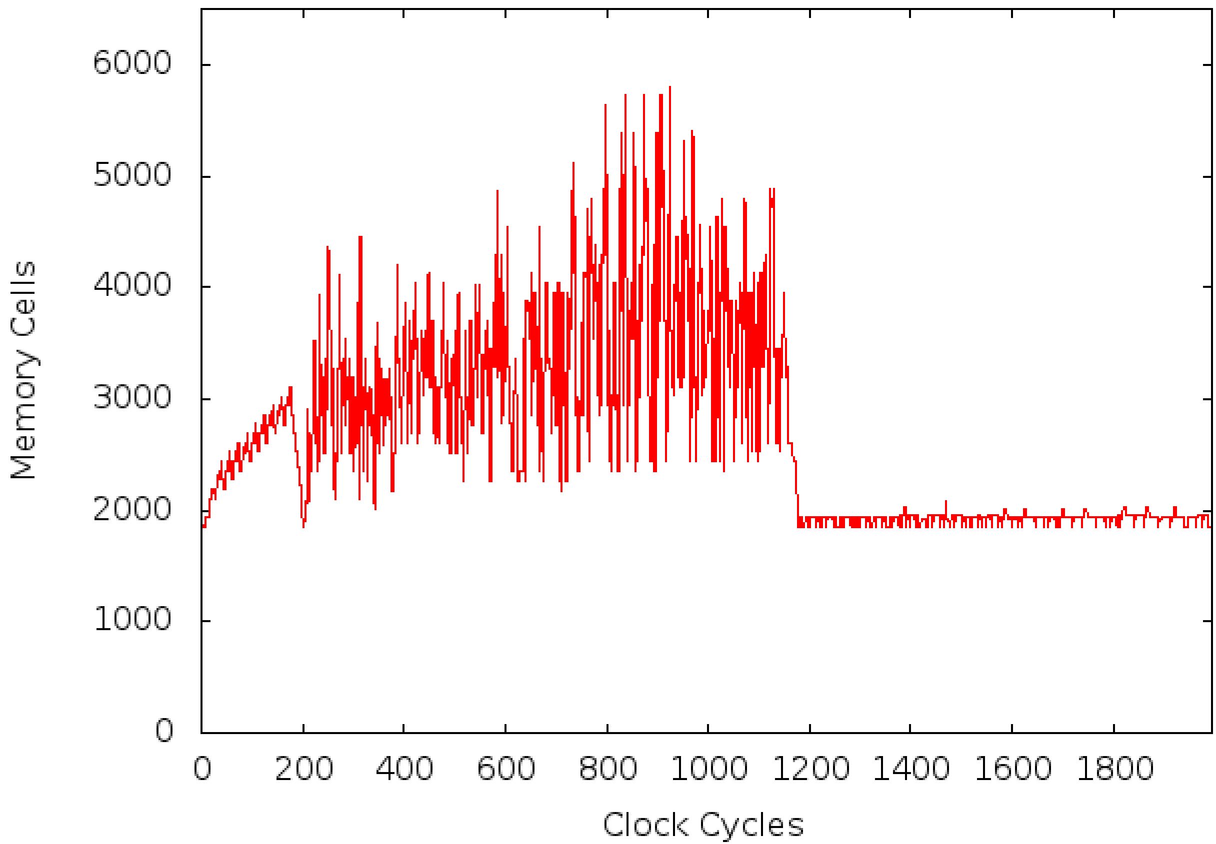
A Simple Macro System:

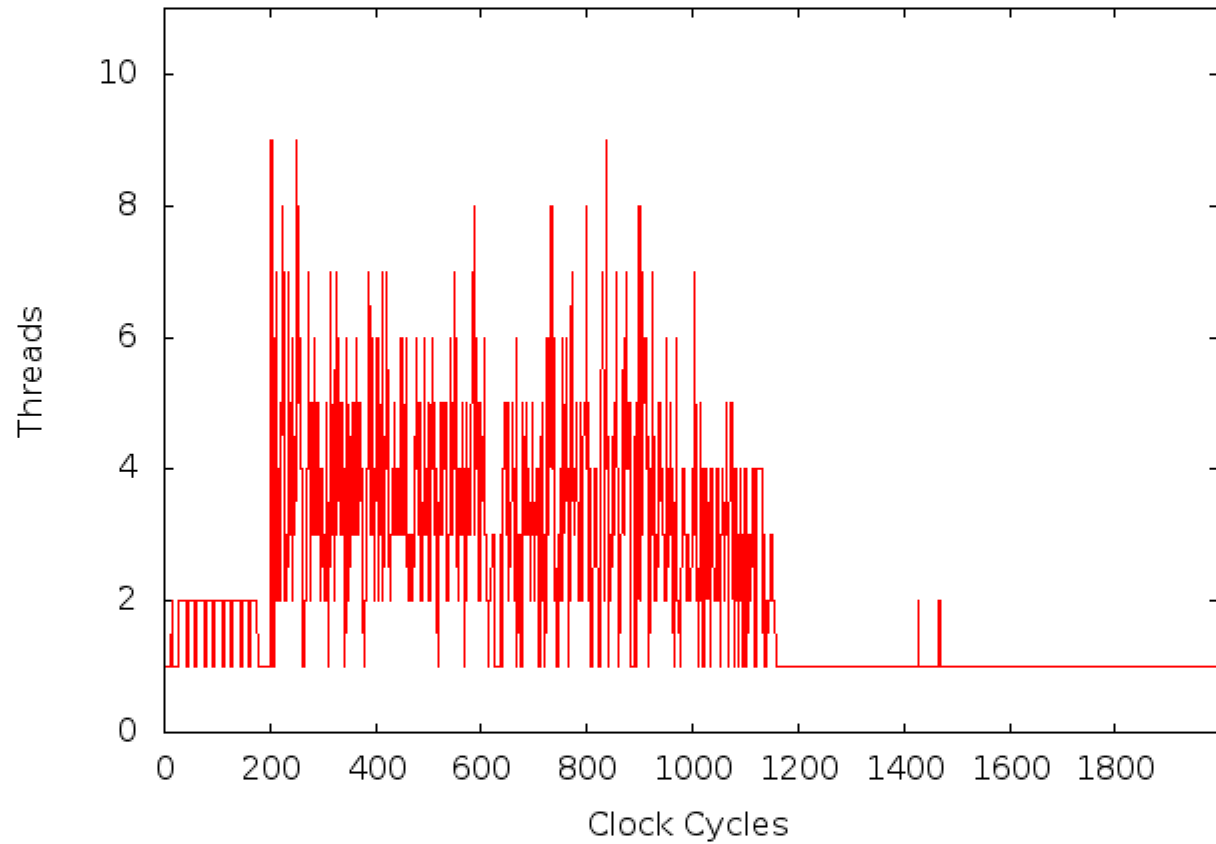
```
(def macro (vau (args e body) env
    ((wrap vau) args e (list env (list e body)))))
```

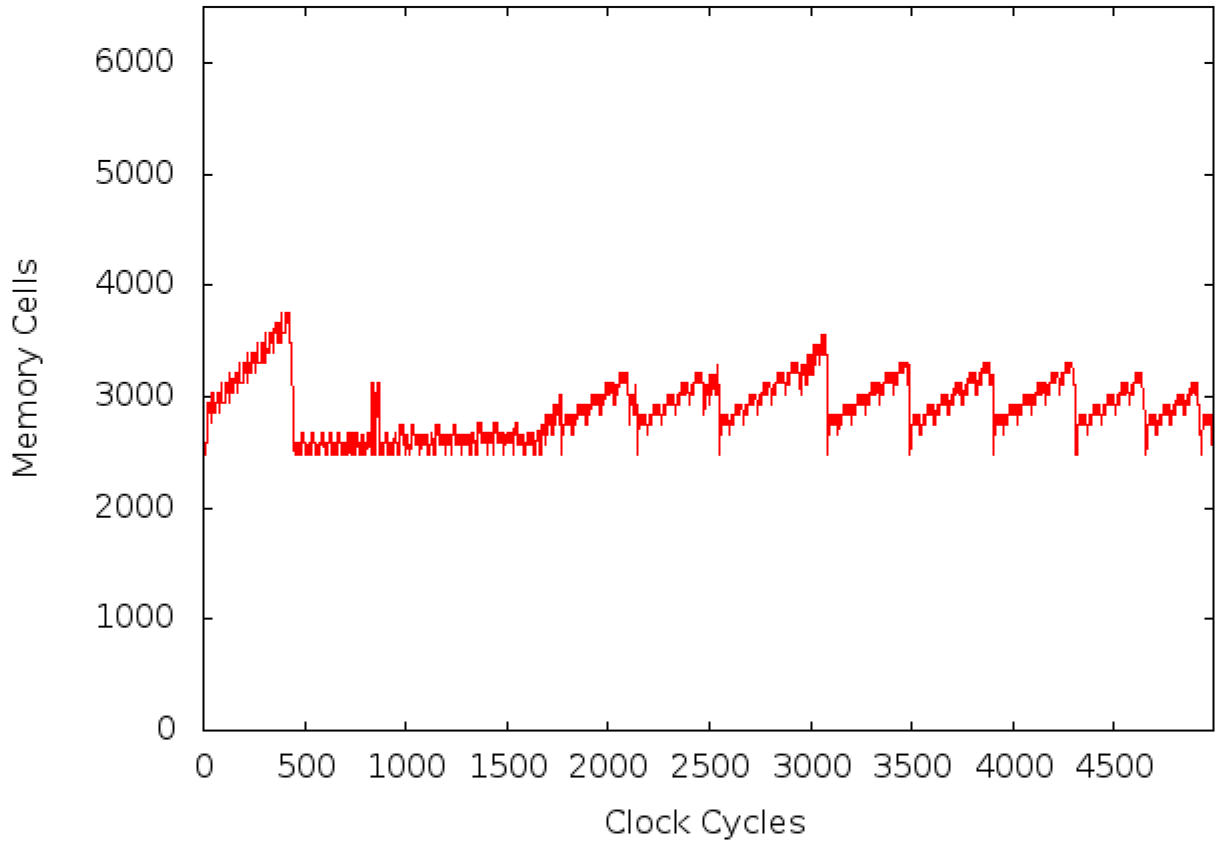
Appendix C: Test Suite

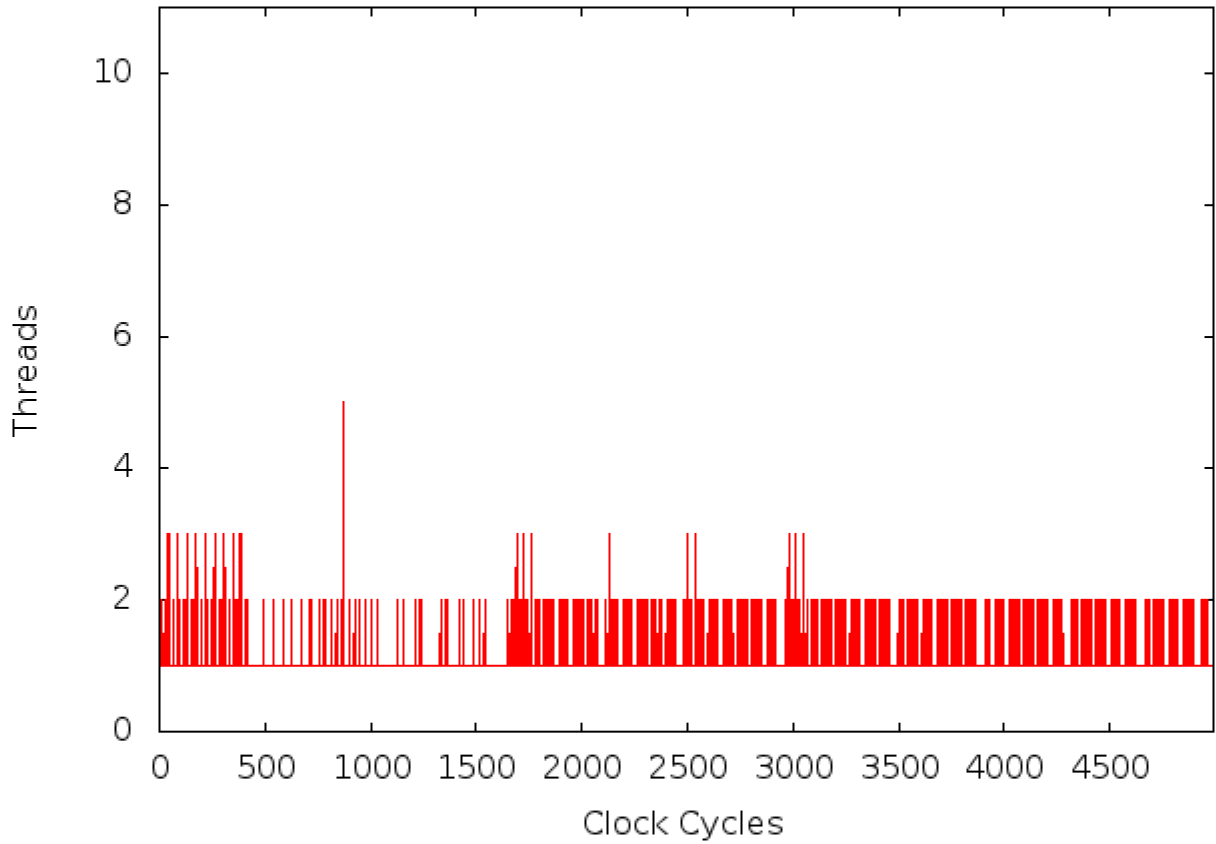
Left-to-Right Tests:	https://github.com/gliese1337/CS598R/blob/master/tests/ltrtest.vrn
Fork/Join Tests:	https://github.com/gliese1337/CS598R/blob/master/tests/snctest.vrn
Lazy Tests:	https://github.com/gliese1337/CS598R/blob/master/tests/lzytest.vrn
Future Tests:	https://github.com/gliese1337/CS598R/blob/master/tests/futtest.vrn

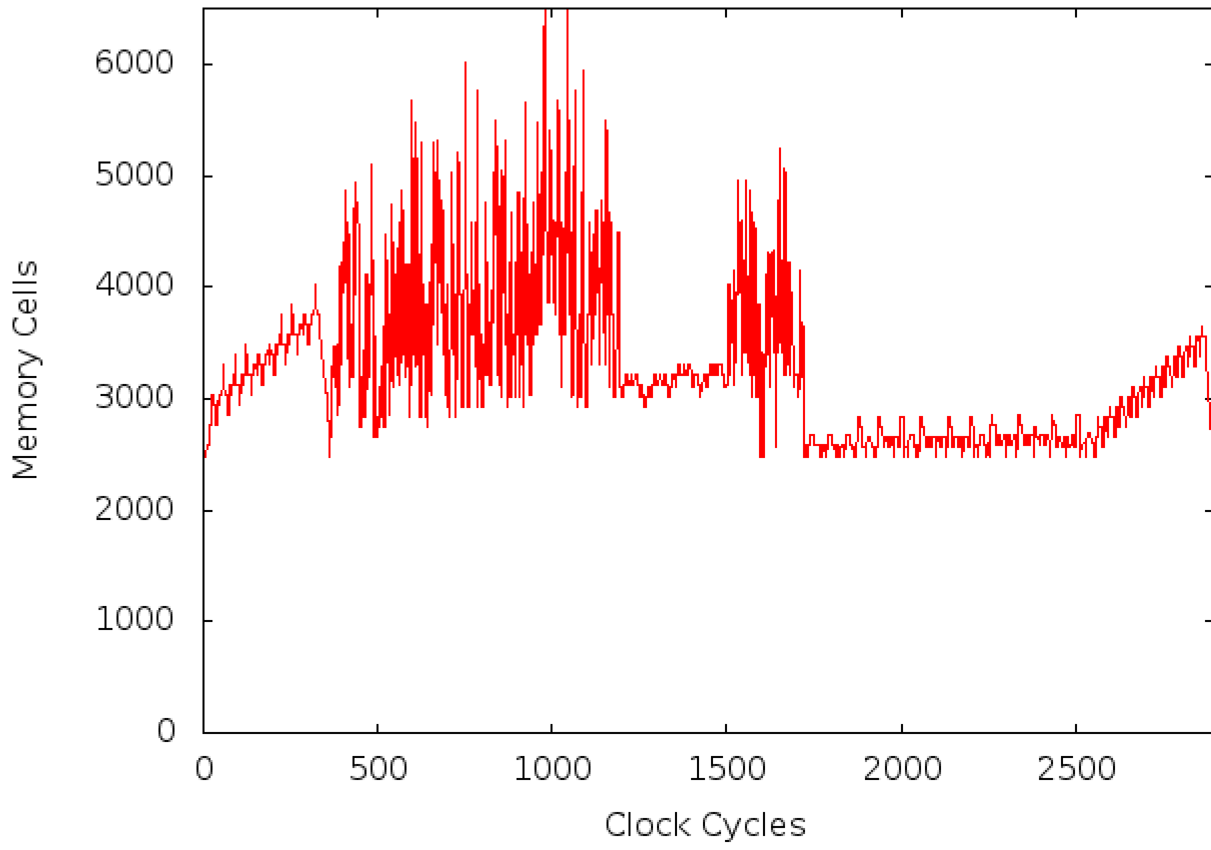
Appendix D: Profiler Charts

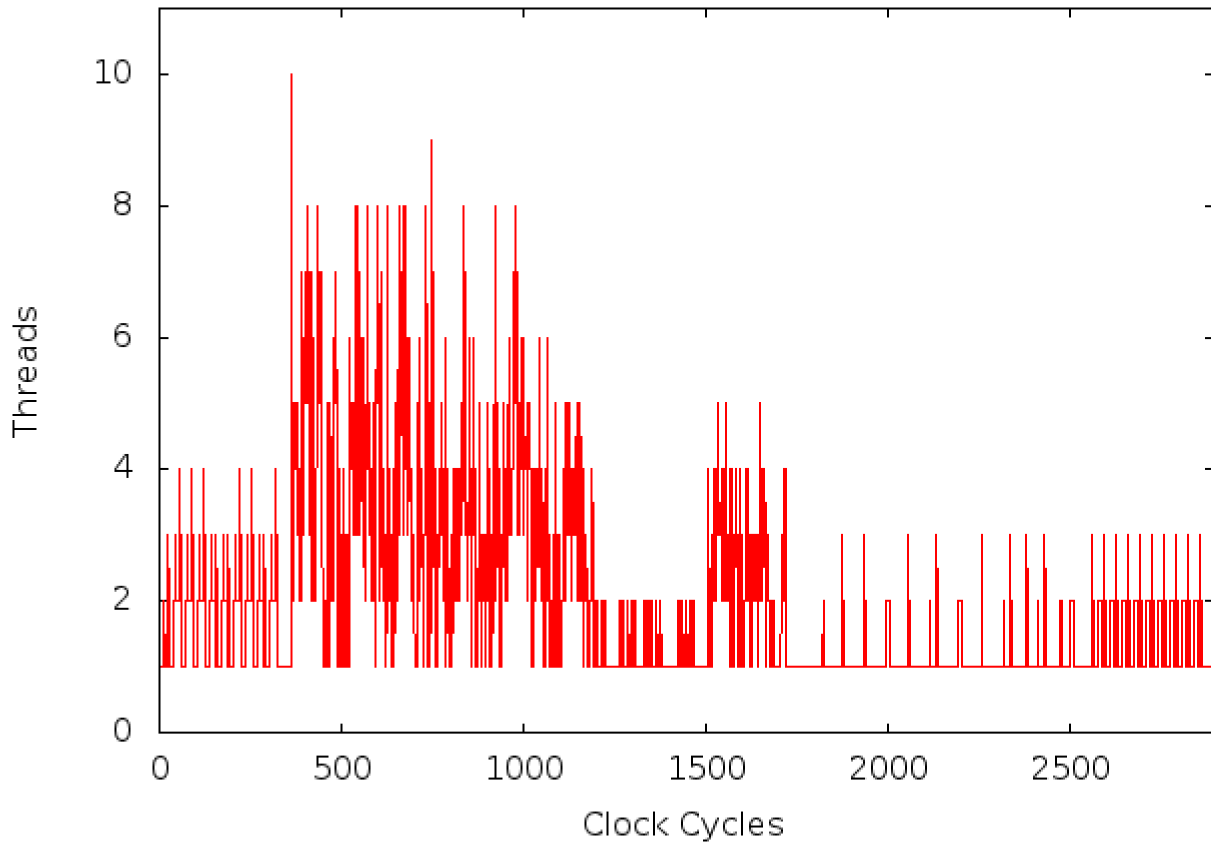


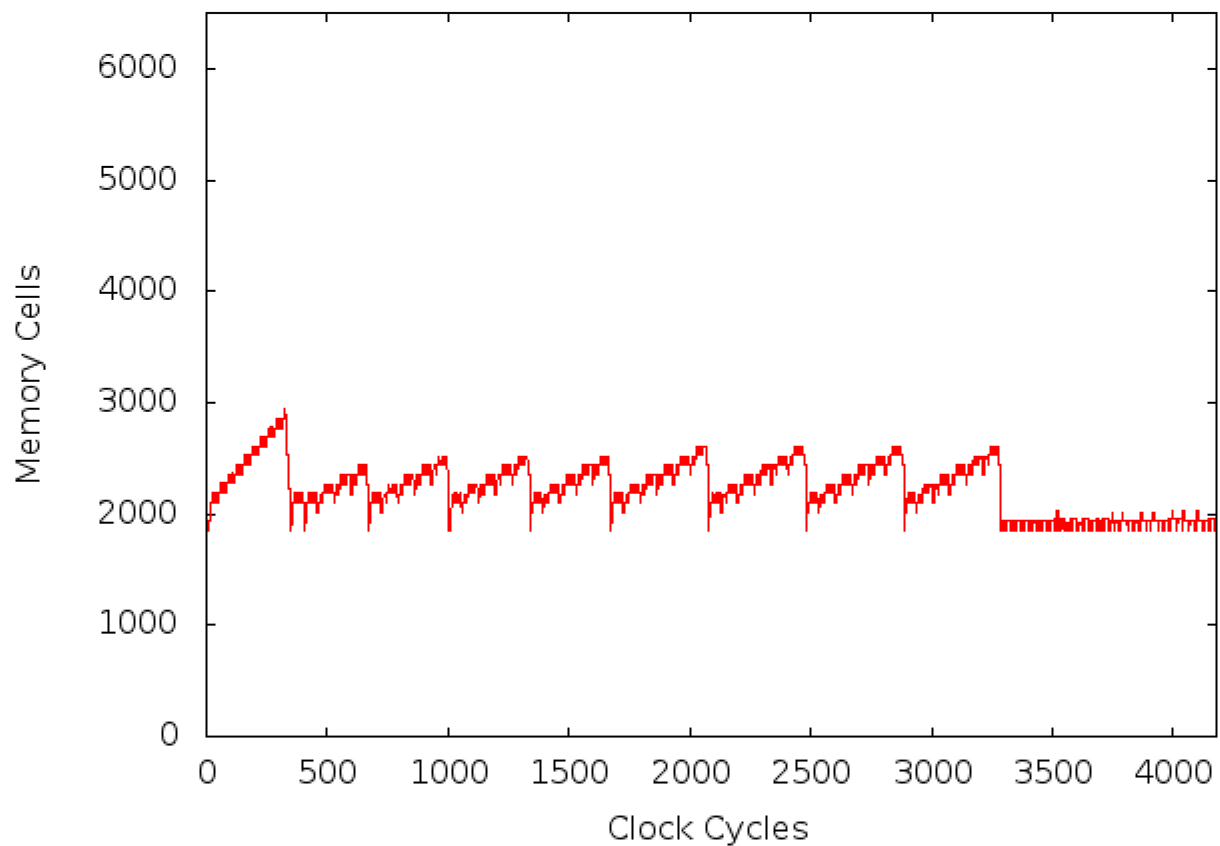



Memory vs. Time for Lazy Evaluation
Memory Cells
0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
Clock Cycles


Threads vs. Time for Lazy Evaluation
Threads
0
2
4
6
8
10
0
500
1000
1500
2000
2500
3000
3500
4000
4500
Clock Cycles


Memory vs. Time for Future Evaluation
Memory Cells
0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
Clock Cycles


Threads vs. Time for Future Evaluation
Threads
0
2
4
6
8
10
0
500
1000
1500
2000
2500
Clock Cycles


Memory vs. Time for Left-to-Right Evaluation
Memory Cells
6000
5000
4000
3000
2000
1000
0
0
500
1000
1500
2000
2500
3000
3500
4000
Clock Cycles