CAL Benchmarking

Purpose:

This document serves a variety of purposes.
It consolidates in a single location a list of the various benchmarks used to test the performance of the CAL runtime. In addition it contains a description of each of the various benchmarks, including notes on specific runtime features exercised by the benchmarks.
The methodology for using the benchmarks to measure CAL performance is also described.
Ongoing information on changes in performance as various optimizations are investigated, as well as notes on the optimizations and their implementations are included.

Benchmarks:

Bonus Rating Benchmarks

testBonusRating30 and testBonusRating200

Description:

These two benchmark functions take a list of String/Double tuples, of length 30 and 200 respectively, which represent a name/ranking pair and separate the tuples into five ranking bands, based on the Double value. The ranking bands are determined by calculating the 5th, 30th, 70th, and 95th percentile values for the collection of ranking values.

Additional Notes:

These two benchmarks are also available in a Haskell version in the file bonusgems.cal.hs.
The `categoriseByBonusRating` gem, which is used by all the bonus rating benchmarks, is a good example of a case where an eta reduction makes a bit difference in performance. By removing the last two arguments to make `categoriseByBonusRating` a function of one argument we can get an order of magnitude increase in performance.

testBonusRating200Eta

Description:

The same as testBonusRating200, except the eta reduction has been applied to the categoriseByBonusRating gem.

Additional Notes:

Bad Regression Gem Benchmarks:

badRegressionGemTest200_200

Description:

These gem take a list of 200 points representing a line and calculate the next 200 points.

Additional Notes:

This gem has a sub-optimal implementation of the function to calculate the lines gradient. The gradient function does a lot of repetitive list creation/manipulation. Also, the badRegressionGem itself is an example of a case where an eta reduction would give a big benefit.

Prime Number Benchmarks:

prime3000
Description:
Determine the 3000th prime number by subscripting the array allPrimes.
Additional Notes:
This benchmark does a fair amount of list manipulation/generation.

getNthPrime5000
Description:
Calculates the 5000th prime using a more procedural algorithm that can be implemented in non-functional languages such as the Crystal Reports formula language.
Additional Notes:

Recursive Constant Space Benchmarks:

foo1000000
Description:
Retrieve the last element of a dynamically generated list of the doubles from 1.0 to 1000000.0.

Additional Notes:
This is a highly recursive benchmark and is good for testing any tail call optimizations. In addition this benchmark should run in a small constant space.

Foreign Function Overhead Benchmarks:

foreignFunctionOverhead1000000
Description:
This benchmark calls a simple foreign function 1 million times. It has a good ratio between calls to the foreign function and CAL functions. This makes it a good choice for benchmarking the overhead of foreign functions.
Additional Notes:
This is a highly recursive benchmark and is good for testing any tail call optimizations. In addition this benchmark should run in a small constant space.

foreignFunctionOverhead10000000
Description:
This is the same as the previous benchmark, except it does 10 million iterations.
Additional Notes:

Primitive Operation Overhead:

primitiveOpOverhead1000000
Description:

This benchmark exercises a primitive operation 1 million times. It is good for testing/measuring the overhead of a primitive operation.
Additional Notes:
This is a highly recursive benchmark and is good for testing any tail call optimizations. In addition this benchmark should run in a small constant space.

primitiveOpOverhead10000000
Description:
This is the same as the previous benchmark, except it does 10 million iterations.
Additional Notes:

primitiveOpOverhead2_1000000
Description:
This is similar to primitiveOpOverhead1000000 except that there is a chain of primitive operations (ex. 1 + 2 – 3) that are in a conditional expression and thus executed strictly.

primitiveOpOverhead2_10000000
Description:
This is the same as the previous benchmark, except it does 10 million iterations.

String Manipulation:

stringManipulation1
Description:
Converts a 200 character string to a JString then back to a string then appends it to another string produced the same way.
Additional Notes:
This benchmark is designed to test String specific optimizations.

Regression and Coverage:

M2.mainM2
Description:
This is the regression testing benchmark. It is included here because it exercises a wide variety of language features.
Additional Notes:
Comparisons of performance on this benchmark across a benchmarking trials are not valid, as M2.mainM2 is continuously being updated as new things are added to the language and prelude. However, it is valid when testing the effect of a specific change to do a before/after comparison.

Data Structure Manipulation:

boxManipulation
Description:
This benchmark creates a fully populated binary tree of Box objects that has n levels, where n will be noted in the timings. The leaf nodes each contain 20 integer boxes containing a list of integers 1 to 20, 1 to 19, etc.

After creating the tree the list of integers in the integer boxes are incremented by 1 (i.e. [1, 2, 3] => [2, 3, 4]).
Finally the integers in all the leaf nodes are summed.
Additional Notes:

boxManipulation2
Description:
This benchmark creates a fully populated binary tree of Box objects that has n levels, where the value of n will be noted in the actual timings. The leaf nodes each contain 20 integer boxes containing a list of integers 1 to 20, 1 to 19, etc.
Because the result of this benchmark is not a simple value (i.e. int, double, Boolean, etc.) it is a good test for conversion of CAL structures to Java structures.
Additional Notes:

Let Variables:

letTest1000000
Description:
This is a recursive benchmark that executes 1 million times. The function declares 4 let variables, one of which is a function, and then uses those variables in a conditional expression.

Application Optimizations:

applyTest4000000
Description:
This is a micro-benchmark designed to test optimizations to fully saturated function applications. A recursive function of arity 3 calls itself 4 million times.

Workbooks:

workbookBenchmark4_10000
Description:
This benchmark creates a workbook that displays a string field and an int field, grouped by the int field with a sum summary on the ints for each group. There are five groups.

Nofib:

digits_of_e1 800
Description:
Calculate the eight hundredth digit of e using continued fractions.

exp3_8
Description:
Calculates 3^8 in a very inefficient way using an abstract data type.

List Traversal:

listTraversalLast
Description:
Take the last element of a 3,000,000 item list five times using Prelude.last.

listTraversalSubscript
Description:
Take the last element of a 3,000,000 item list five times using Prelude.subscript.

SimpleRolap2:

simpleRolap2
Description:
Run SimpleRolap2.verifySimpleCube1 for a cube size of 2,700,000.

Let variables:

Some benchmarks of let variable use.
The 'a' version is written using lets and the 'b' version is written with the let variables manually in-lined. The purpose is to compare the performance of how we generate let variables compared to directly in-lining the let variable definition.

Currently we will always lift the definition of the let variable into its own java method. If it is a single use let variable we inline the call to the definition function at the point the variable is referenced. We try to be smart about compiling the body of the let definition function based on the usage of the let variable and the nature of the definition so that we compile strictly, returning an unboxed value, etc.

letVarBenchmark_1_a , letVarBenchmark_1_b
Description:
The let variable is single use in the main function body and the usage is lazy. The definition is such that laziness can be ignored.

letVarBenchmark_2_a, letVarBenchmark_2_b
Description:
This is the same benchmark as letVarBenchmark_1_a/b except that the plinging of a function argument causes the single use of the let variable to be strict.

letVarBenchmark_3_a, letVarBenchmark_3_b
Description:
The let variable is such that we cannot ignore laziness in the definition. It is single use in the main function body and the usage is lazy.

letVarBenchmark_4_a, letVarBenchmark_4_b
Description:

The same as letVarBenchmark_3_a/b except that plinging of a function argument causes the single usage of the let variable to be strict.

letVarBenchmark_5_a, letVarBenchmark_5_b
Description:
In this case the definition function for x is compiled strictly because the definition of x is such that laziness can be ignored.
The body of the z definition function is also compiled strictly because it is a single use let variable which is used strictly.
The difference between the 'a' and 'b' cases should be solely due to the overhead in calling the z definition function.

Code:

```
letVarBenchmark_5_a :: Int -> Int;
letVarBenchmark_5_a n =
   let
      helper :: Int -> Int;
      helper !i =
            if (i < n) then
               let
                  x :: Int;
                  x = 1 + 2 ;
                  z ::Int;
                  z = x + x;
               in
                  helper (i + z)
            else n;
   in
       helper 0;

letVarBenchmark_5_b :: Int -> Int;
letVarBenchmark_5_b n =
   let
      helper :: Int -> Int;
      helper !i =
            if (i < n) then
               let
                  x :: Int;
                  x = 1 + 2;
               in
                  helper (i + (x + x))
            else n;
   in
       helper 0;
```

letVarBenchmark_6_a, letVarBenchmark_6_b

Description:
In this case x is used strictly/unboxed in the definition of z.
The body of the z definition function is compiled strictly because z is single use in a strict context. However, we don't currently determine that all code paths in the z definition evaluate/unbox x. As a result the x definition function is compiled lazily and we pass a boxed value to the z definition function. The definition of z is in-lined in the 'b' version.

Code:

```
letVarBenchmark_6_a :: Int -> Int;
letVarBenchmark_6_a n =
   let
      helper :: Int -> Int;
      helper !i =
            if (i < n) then
               let
                  x :: Int;
                  x = simpleIntFunctionStrict 1 1;
                  z :: Int;
                  z = x + x;
               in
                  helper (i + z)
            else n;
   in
       helper 0;

letVarBenchmark_6_b :: Int -> Int;
letVarBenchmark_6_b n =
   let
      helper :: Int -> Int;
      helper !i =
            if (i < n) then
               let
                  x :: Int;
                  x = simpleIntFunctionStrict 1 1;
               in
                  helper (i + (x + x))
            else n;
   in
       helper 0;
```

letVarBenchmark_7_a, letVarBenchmark_7_b
Description:
There is a significant difference in performance between letVarBenchmark_7_a and letVarBenchmark_7_b. Even though the use of x in z is strict/unboxed we aren't currently doing the analysis to determine whether this usage occurs in all code paths of the definition of z. As a result we

have to generate a lazy graph for x in the x definition function and pass this boxed value to the z definition function.
Also, both the x and z let variables are single use. So in the 'b' case the definition of both variables is in-lined.

Code:

```
letVarBenchmark_7_a :: Int -> Int;
letVarBenchmark_7_a n =
   let
      helper !i =
            if (i < n) then
               let
                  x = simpleIntFunctionStrict 1 1;
                  z = x + 1;
               in
                  helper (i + z)
            else n;
   in
       helper 0;

letVarBenchmark_7_b :: Int -> Int;
letVarBenchmark_7_b n =
   let
      helper !i =
            if (i < n) then
                  helper (i + ((simpleIntFunctionStrict 1 1) + 1))
            else n;
   in
       helper 0;
```

Others:

averageDynamicList
Description:
Calculates the average of a list of 2,000,000 doubles, where the list is dynamically generated. I.e. average(upFromTo 1.0 2000000.0).

averageExpandedList
Description:
Caculates the average of a list of 2,000,000 doubles, where the list of doubles is fully realized in memory. I.e. average (expandedDoubleList 2000000).

benchmarkPartialApplications
Description:

For a set of functions of arity one through ten; Each possible partial application of each function is used to transform a list which is averaged.

Obsolete Benchmarks:

These are benchmarks that no longer exist. The descriptions are kept here for the purpose of interpreting older benchmark results.

testBonusRating60
Description:
This benchmark function take a list of String/Double tuples, of length 30, which represent a name/ranking pair and separate the tuples into five ranking bands, based on the Double value. The ranking bands are determined by calculating the 5^{th}, 30^{th}, 70^{th}, and 95^{th} percentile values for the collection of ranking values.
Additional Notes:
Obsolete as of Jan. 27^{th} 2003

testBonusRatingLayout30 and testBonusRatingLayout60
Description:
The previous testBonusRating benchmark, plus laying out the results into a Container hierarchy.
Additional Notes:
Obsolete as of Jan. 27^{th} 2003

testBonusRatingLayoutAndDisplay30 and testBonusRatingLayoutAndDisplay60
Description:
The previous testBonusRatingLayout benchmark, plus converting the container hierarchy to XML and writing it out to a disk file, then invoking the viewer app.
Additional Notes:
Obsolete as of Jan. 27^{th} 2003

Bad Regression Gem Benchmarks:

badRegressionGemTest20_20 and badRegressionGemTest40_20
Description:
These two gems take a list of 20 points representing a line and calculate the next 20 and 40 points respectively.
Additional Notes:
Obsolete as of Jan. 27^{th} 2003

badRegressionGemTest20_40 and badRegressionGemTest40_20
Description:
These two gems take a list of 40 points representing a line and calculate the next 20 and 40 points respectively.
Additional Notes:
Obsolete as of Jan. 27^{th} 2003

Prime Number Benchmarks:

prime200 and prime400
Description:
Determine the 200th and 400th prime respectively.
Additional Notes:
Obsolete as of Jan. 27th 2003

Recursive Constant Space Benchmarks:

foo10000 and foo20000
Description:
Retrieve the last element of a dynamically generated list of the letter a. The list is of length 10000 and 20000 respectively.
Additional Notes:
Obsolete as of Jan. 27th 2003

Foreign Function Overhead Benchmarks:

foreignFunctionOverhead
Description:
This benchmark calls a simple foreign function 100000 times. It has a good ratio between calls to the foreign function and CAL functions. This makes it a good choice for benchmarking the overhead of foreign functions.
Additional Notes:
Thi Obsolete as of Jan. 27th 2003

Primitive Operation Overhead:

primitiveOpOverhead
Description:
This benchmark exercises a primitive operation 100000 times. It is good for testing/measuring the overhead of a primitive operation.
Additional Notes:
Obsolete as of Jan. 27th 2003

Benchmark Methodology:

The benchmarks are in the Benchmarks module. The Benchmarks module directly relies on gems in the Prelude, BonusRatings, M1, and GemScopeTestModule modules.

The individual benchmarks can be run using either ICE or the GemCutter. It is recommended that ICE be used for two reasons. 1) ICE gives more detailed performance information. 2) The ':pt' command in ICE will automatically generate averaged performance information across several runs of a benchmark.

To run a set of benchmarks ICE can run a benchmark script using the ':bs' command. A benchmark script file contains a list of benchmarks to run. ICE will run all the benchmarks in the list using the ':pt' command then display the performance info.

Benchmark Results:

Nov. 8, 2002: Initial baseline performance.

```
testBonusRating30
time = 9484, instructions = 3793021, fips = 1058602

testBonusRating60
time = 42231, instructions = 17054317, fips = 1063703

testBonusRatingLayout30
time = 9834, instructions = 3918717, fips = 398466

testBonusRatingLayout60
time = 42972, instructions = 17285351, fips = 402227

badRegressionGemTest20_20
time = 2329, instructions = 1296734, fips = 556587

badRegressionGemTest40_20
time = 4645, instructions = 2591214, fips = 557782

badRegressionGemTest20_40
time = 4625, instructions = 2571014, fips = 555867

badRegressionGemTest40_40
time = 9169, instructions = 5137494, fips = 560268

prime200
time = 2086, instructions = 943554, fips = 452692

prime400
time = 7748, instructions = 3467004, fips = 447525

foo10000
time = 1566, instructions = 530047, fips = 339876

foo20000
time = 3169, instructions = 1060047, fips = 335019

foreignFunctionOverhead
time = 5485, instructions = 2100029, fips = 382865

primitiveOpOverhead
```



```
time = 5530, instructions = 2400029, fips = 434074

stringManipulation1
time = 180, instructions = 72471, fips = 343809

m2.mainM2
Time = 942, instructions = 416934, fips = 445626
```

Nov. 8, 2002: Type specific value nodes and packed strings.

Introduction of specific node types for primitive values (i.e. char, int, double). Also a packed representation of literal strings was added and smart dispatch applied to the functions stringToJString, jStringToString, and append to take advantage of the packed representation.

```
testBonusRating30
time = 9440, instructions = 3789655, fips = 864673

testBonusRating60
time = 42285, instructions = 17047315, fips = 1307952

testBonusRatingLayout30
time = 9744, instructions = 3866127, fips = 396779

testBonusRatingLayout60
time = 42860, instructions = 17187177, fips = 401001

badRegressionGemTest20_20
time = 2345, instructions = 1296734, fips = 552729

badRegressionGemTest40_20
time = 4665, instructions = 2591214, fips = 555395

badRegressionGemTest20_40
time = 4629, instructions = 2571014, fips = 555391

badRegressionGemTest40_40
time = 9252, instructions = 5137494, fips = 555260

prime200
time = 2046, instructions = 943554, fips = 461000

prime400
time = 7718, instructions = 3467004, fips = 449276

foo10000
time = 1564, instructions = 530047, fips = 339940

foo20000
time = 3178, instructions = 1060047, fips = 334135

foreignFunctionOverhead
time = 5842, instructions = 2100029, fips = 359465

primitiveOpOverhead
```



```
time = 5864, instructions = 2400029, fips = 409294

stringManipulation1
time = 10, instructions = 50, fips = 4600

m2.mainM2
time = 778, instructions = 343659, fips = 441471
```

Changes:

The only benchmarks that show a noticeable improvement are stringManipulation1 and M2.mainM2 with performance increases of 94% and 17% respectively. Unfortunately the performance change seen in the stringManipulation1 benchmark doesn't carry over to more general code as the packed representation of a string is quickly lost. However, there has been some discussion as to ways to allow packed representations of dynamically generated strings. This would, if it didn't introduce too much overhead, give corresponding performance increases to a greater variety of string manipulations.

Dec. 11, 2002: New compilation schemes based on original Peyton-Jones text.

The new compilation schemes are smarter about strictness vs. laziness and try to be optimal in a wider variety of cases.

Baseline using the current implementation:

	Time (ms)	Instructions	FIPs
TestBonusRating30	8544	3 843 773	677 156
TestBonusRating60	38466	17 302 733	1 078 183
TestBonusRatingLayout30	Failed		
TestBonusRatingLayout60	Failed		
BadRegressionGemTest20_20	2181	1 307 251	599 103
BadRegressionGemTest40_20	4315	2 612 211	608 105
BadRegressionGemTest20_40	4292	2 591 971	604 678
BadRegressionGemTest40_40	8618	5 179 331	600 997
Prime200	1919	946 008	493 693
Prime400	7101	3 472 498	489 018
Foo10000	1508	530 047	352 719
Foo20000	2969	1 060 047	357 741
ForeignFunctionOverhead	5046	2 100 029	416 243
PrimitiveOpOverhead	4985	2 400 029	481 384
StringManipulation1	0	50	---
M2.mainM2	783	365 596	467 086

Total Time: 89 727 ms

Benchmarks run using new compilation schemes.

	Time (ms)	Instructions	FIPs
TestBonusRating30	3113	3280305	1295945
TestBonusRating60	13752	14764845	2050221
TestBonusRatingLayout30	Failed		

TestBonusRatingLayout60	Failed		
BadRegressionGemTest20_20	1029	1176616	1143414
BadRegressionGemTest40_20	2030	2351156	1157949
BadRegressionGemTest20_40	2025	2333516	1152098
BadRegressionGemTest40_40	4018	4662856	1160370
Prime200	816	826302	1017414
Prime400	3048	3026297	993996
Foo10000	509	450045	894571
Foo20000	976	900045	925550
ForeignFunctionOverhead	1812	1900027	1049689
PrimitiveOpOverhead	1810	2200027	1221894
StringManipulation1	85	65836	976017
M2.mainM2	409	379465	932129

Total Time: 35 432

The new compilation schemes deliver significance performance benefits for all the benchmarks except stringManipulation1. However, the new implementation doesn't have the optimization to represent strings in a packed (i.e. native java) format. It is expected that with this optimization in place this one discrepancy will vanish.

Jan. 27th 2003: Comparisons between machine versions before/after space leak fixes and tail call optimizations.

Some benchmarks comparing various machine versions. The current state of the g-machine, with and without the memory leak (indirection chain) problem fixed, the g2 version with the initial version of the tail call optimization, and the g2 version with the dispatch state transitions optimized. (Note: there are still further optimizations to be done with regards to tail calls but this is a good point to get some numbers).

Benchmark	Original (g)	With memory leak fix	Initial tail call changes	2nd Tail call changes
TestBonusRating200	102217	104310	108356	102337
TestBonusRating200 Eta	681	701	721	671
BadRegressionGemTest	53276	56962	52826	53367
Prime3000	99433	114695	117849	112181
Foo1000000	64293	62499	49722	49041
ForeignFunction1000000	67587	9584	8793	8843
PrimitiveOp1000000	60287	9824	8863	8713
StringManipulation	60	61	50	50
M2.mainM2	380	360	260	261
Total	448214	358996	34744	335464

The results from these benchmarks are interesting.

The first thing that became apparent, and which I confirmed by further investigation, is that the fix for the memory leak (i.e. eliminating indirection chains) robs the tail call optimization of a lot of its punch. I investigated and determined that the tail call optimizations also prevent the memory leak in recursive cases, since updating the root is put off until all the recursive cycles have executed rather than updating each cycle.

It was interesting to note that the non-recursive benchmarks actually slowed down with the memory leak fix and first implementation of the tail call optimization and were simply restored to their original performance by the 2nd tail call change.

Also interesting is the fact that the cumulative changes with the memory leak fix and the tail call optimization is not always positive. As can be seen there is a negligible change in TestBonusRating200 and BadRegressionGemTest. The highly recursive benchmarks (foo1000000, ForeignFunction, and PrimitiveOp) all showed strong improvement. StringManipulation and M2.mainM2 both showed improvement, but are so short that this may be a false indication of improvement. The big surprise was the Prime3000 benchmark, which actually slowed down by almost 13%. The slow down in the primes benchmark is very disappointing and I'll be investigating to see if I can determine what it is about this benchmark that reacts so negatively to the changes.

Benchmark	% improvement overall
TestBonusRating200	(0.11)
TestBonusRating200 Eta	1.5
BadRegressionGemTest	(0.17)
Prime3000	(12.8)
Foo1000000	20.9
ForeignFunction1000000	86.9
PrimitiveOp1000000	85.5
StringManipulation	16.7
M2.mainM2	31.3
Total	25

Overall I'm quite disappointed in the overall performance benefits from the tail call optimization to date. Once the memory leak is fixed the tail call optimization makes no noticeable difference in our two benchmarks that are the closes to 'real world' code (i.e. TestBonusRating200 and BadRegressionGemTest). One thing that has become apparent is that we need some more benchmarks that test the kind of things that are actually being done with CAL. While the performance improvement in the recursive benchmarks is large (and gratifying) it doesn't help much if unless we can convince end user to write a lot of recursive gems.

There are two things left to do with the tail call optimization:

1) The state transition for partial applications can be further optimized.
2) The keyhole optimizer can be used to optimize the occurrences of the dispatch instruction.

I don't currently have a good feel for how much difference these two changes will make. The 1st depends on how often we get a partial application as a tail call. The 2nd gains much of its usefulness by eliminating the need for the dispatch instruction to do case analysis of the type of node on top of the stack. In our case polymorphic dispatch is doing this already. However, I still think they are worth doing, if for no other reasons than completeness and consistency.

Space Usage

The effect of the tail call optimization on space usage is, generally, to reduce the total number of nodes allocated and the maximum number of simultaneously existing nodes. However, given the efficiency with which java seems to release/reallocate Node objects this change in memory usage probably doesn't give much of a performance boost.

One very interesting thing to note is that the primes benchmark actually shows an increase in the total number of nodes and the maximum simultaneous number of nodes. This could account for the decrease in performance, especially when finding large primes. This will need further investigation.

testBonusRating30

	G	G2	% difference
Total # nodes allocated	10931	10322	(5.6)
Max simultaneous	353	314	(11)
Max stack size	82	82	

sum (GemScopeTestModule.badRegressionGem 20 GemScopeTestModule.regressionLine 20) 20.0)

	G	G2	% difference
Total # nodes allocated	312265	297692	(4.6)
Max simultaneous	232	229	(1.2)
Max stack size	62	62	

subscript allPrimes 100

	G	G2	% difference
Total # nodes allocated	52426	56984	8.6
Max simultaneous	724	731	0.9
Max stack size	344	344	

last (take 1000 (upFrom 1.0))

	G	G2	% difference
Total # nodes allocated	30012	27013	(9.9)
Max simultaneous	37	36	(2.7)
Max stack size	19	19	

Jan. 30th, 20003: Introduction of a continuation stack to remove the use of the I_Cont instruction.

Profiling showed that a few classes were responsible for a (potentially) disproportionately large number of allocations in the course of running a program. One of these classes is the I_Cont instruction. Instances of the I_Cont instruction are created at runtime and used to bring execution back to a particular point after branching to a different code object. An alternative to chaining the new code offset object back to the current code offset object via an I_Cont is to use a continuation stack, where the current code offset is pushed on to the stack before

branching. When a code offset reaches the end of the associated code the 'continuation' is popped off of the stack and execution continues. This continuation stack has similar behavior to the program stack in that it needs to be segmented, cleared, and restored when dump items are pushed and popped.
An added benefit of removing the use of I_Cont instructions is that the codd offset object becomes smaller as it no longer needs to hold a reference to this chaining instruction.

Benchmark	With continuation instruction	Without continuation instruction
TestBonusRating200	130050	135070
TestBonusRating200 Eta	870	891
BadRegressionGemTest	67814	71282
Prime3000	144858	145152
Foo1000000	65637	86190
ForeignFunction1000000	12010	14347
PrimitiveOp1000000	10812	12601
StringManipulation	63	69
M2.mainM2	330	317
Total	432444	465919

Benchmark	% improvement overall
TestBonusRating200	(3.8)
TestBonusRating200 Eta	(2.4)
BadRegressionGemTest	(5.1)
Prime3000	(0.2)
Foo1000000	(31)
ForeignFunction1000000	(19)
PrimitiveOp1000000	(16)
StringManipulation	(9.5)
M2.mainM2	3.9
Total	(7.7)

As an optimization these changes are a complete failure. Even though the changes eliminate a large number of object allocations the benchmarks actually degrade in performance rather than improving. The conclusion that can be drawn is allocating a lot of short lived I_Cont objects is handled very efficiently by the JVM memory management. However, creating a stack and pushing/popping a corresponding number of already existing objects is much more expensive.

Feb. 12th, 2003: Changes to eliminate continuation instructions, optimize pushing global nodes, and speed up unwinding.

Profiling showed that a few classes were responsible for a (potentially) disproportionately large number of allocations in the course of running a program. One of these classes is the I_Cont instruction. Instances of the I_Cont instruction are created at runtime and used to bring execution back to a particular point after branching to a different code object. Examination showed that I_Cont instructions were used in two different scenarios: 1) a conditional (if or switch) branches to a different instruction sequence but wants execution to resume at the current point when the branch has finished, 2) when a client is unwinding a WHNF result generated code is inserted into the current code sequence.

Times in ms:

Benchmark	Baseline	I_Cond uses Jump	Prelude.if uses Jump	Switches Use Jumps	Link code to PushGlobal	Unwind
TestBonusRating200	106723	103659	102758	97800	91371	85043
TestBonusRating200 Eta	701	691	691	671	601	551
BadRegressionGemTest	56101	54728	53898	53427	49652	46627
Prime3000	131008	132801	132691	131279	110589	89649
Foo1000000	62470	62210	61278	49090	54248	43322
ForeignFunction1000000	9864	8642	8532	9053	8061	7711
PrimitiveOp1000000	9184	8192	7972	8482	7671	7251
StringManipulation	70	60	70	60	50	30
M2.mainM2	390	361	340	330	281	240
Total	376511	371344	368230	350192	322524	280424

The first step in getting rid of continuation instructions in the first scenario was to change the implementation of the I_Cond (conditional) instruction to use a jump rather than branching and returning. This gave a small improvement of only 1.4% over the benchmark suite.
The next step was to change the code generated for the function Prelude.if to directly use jump instructions. This generated an improvement of 0.8% in the benchmark suite for a cumulative improvement of 2.1% over the baseline figures.
The final step was to modify switches to use jumps rather than an n-way branch. This resulted in an improvement of 4.9%, for a cumulative improvement of 7% over the baseline results.

Another object that was being allocated excessively was the code-offset. Eliminating the use of this class involves multiple steps and isn't yet complete.
However, the first step was to eliminate the code re-write optimizations. Without the code re-writing code objects can be treated as immutable, which simplifies the use of offsets. Unfortunately the code re-write optimizations gave a large performance boost by speeding up retrieval of the code for global supercombinators from the program cache. The solution was to eliminate the need for retrieval via the cache by doing a linking step after code has been generated for all supercombinators. This means that an I_PushGlobal instruction doesn't cause a fetch of the appropriate code from the program/program-cache, nor does it mean creating a copy of the code (since the code is immutable). An added benefit was the ability to use a single instance of an NGlobal node for all instances of a given supercombinator.
The result of these changes was another 7.9% improvement, bringing the improvement over the baseline to 14.3%.

Having made the previous changes it was possible to re-work the scheme by which a client of the runtime interactively unwinds WHNF results. This resulted in the complete elimination of the continuation instruction. It also eliminated the need to dynamically generate 'unwind code' on each unwind cycle.
The result is another 13% improvement on the benchmark suite, for a total improvement of 25.5% over the baseline.

March 24th, 2003: Eliminate CodeOffset class and cut down on polymorphic function calls and casting.

The Executor was modified to use a 'currentCode' member and an integer offset into the current code rather than a CodeOffset object. This allowed for the elimination of the CodeOffset class saving the allocation of a CodeOffset object every time a function is entered.
The fetch/execute loop in the executor was modified. Instead of calling a virtual function of the instruction, which then calls back into the executor, the fetch/execute loop looks at a tag in the instruction object to choose the appropriate case from a switch statement and do the state transition directly.
The Instruction base class was modified to contain all the necessary data members for the different derived classes. These members, and their access functions, were made final. This eliminated the need to cast the instruction object in order to access data members.
In addition the Code class was made machine specific so that the fetch/execute loop didn't need to continuously cast from an InstructionBase reference to an Instruction reference.
The minimum improvement in the benchmarks was 11.78% on the badRegressionGemTest benchmark. The best improvement was 17.98% on the foreignFunctionOverhead benchmark. The prime3000 benchmark also showed good improvement with an improvement of 17.42%.

Benchmark	Baseline	Optimized	% Improvement
TestBonusRating200	108486	93014	14.26
BadRegressionGemTest	58064	51223	11.78
Prime3000	108486	89759	17.42
Foo1000000	66946	57883	13.54
ForeignFunctionOverhead	10305	8452	17.98
PrimitiveOpOverhead	9203	7641	16.97
BoxManipulation	65714	57923	11.86
Total	427885	365895	14.83 (Average of above)

April 2nd, 2003: Addition of new basic type, 'String'.

A new basic type, 'String', was added. The internal representation is a java string object, and the foreign function interface directly passes CAL String objects over as a java String object. The '++' operator was changed to mean 'appendStrings' and literal strings in CAL code are compiled as type String.
The various existing CAL scripts were updated to use String vs. [Char] appropriately.

Benchmark	Baseline	String	% Improvement
TestBonusRating200	73906	73876	0.04
BadRegressionGemTest	40869	40768	0.24
Prime3000	76280	73857	3.17
Foo1000000	47858	46907	1.98
ForeignFunctionOverhead	6620	6550	1.05
PrimitiveOpOverhead	6239	6188	0.82
BoxManipulation	47248	46127	2.37
Total	299020	294273	1.38 (Average of above)

Changes in performance to the benchmarks were negligible. This was expected as none of the benchmarks do any String manipulation.

Tests on simple workbooks showing an attributed list of values showed an improvement of 14% - 23%.

April 16th, 2003: Changes to unwind mechanism.

A bug was discovered with the client-side unwind mechanism.
If the following data declarations were added to the prelude:

```
data Foo a b = Bar (Baz a) (Cuz a) a;
data Baz a = Baz a;
data Cuz a = Coo;

mtest :: Boolean;
public mtest = False;
```

Then the expression:
Bar (Baz (mtest)) Coo (mtest)
would crash ICE with an ArrayIndexOutOfBounds exception. This was caused because the node for the result of mtest appeared twice in the unwinding process. The first time it was encountered as the third child of the root it had the associated output listener set. However, before it was unwound it was encountered again in the process of unwinding the first child of the root '(Baz (mtest))' and had the output listener set again, overwriting the first output listener. Thus the result node of mtest tried to send its value to the same output listener twice.

Changing the node class to hold a stack of output listeners that were popped as they were used proved to be unacceptably expensive in terms of performance. Requiring the unwind listener to keep track of the output listeners associated with parts of the result tree made client code significantly more complex and error prone.

Ultimately the solution adopted was to use the runtimes stack to keep track of the output listeners. Thus, when unwinding, there is always a pair of objects on the stack, the output listener followed by the associated root. Running boxManipulation2, the benchmark which involves extensive unwinding three times gave the following values.

Old Unwind Code	New Unwind Code	% difference
34460	34229	0.67
32357	32297	0.18
31926	32276	(1.09)

The performance change from before the bug fix to after the bug fix is negligible.

July 3, 2003: Bug fixes to compilation of overloading code.

	Pre-compilation changes. (May 30th build)	Post compilation changes. (June 27th build)	Change
TestBonusRating200	73286	27991	61.8%
BadRegressionGemTest200_200	40288	40659	(0.9)%
Foo1000000	38986	38445	1.3%
ForeignFunctionOverhead1000000	7020	7041	(0.3)%
PrimitiveOpOverhead1000000	6519	6439	1.2%
Prime3000	72355	74417	(2.8)%

GetNthPrime5000	57243	57032	(0.4)%
BoxManipulation	46768	46247	1.1%
BoxManipulation2	22552	22562	(0.04)%

These results show that the compilation changes had a negligible effect on all the benchmarks except for testBonusRating200. This benchmark shows a dramatic improvement of 61%.
One scenario that the changes fixed was generating code that caused and expression that evaluated to a value (e.g. Double, Integer) to be applied as a function. The g-machine would unwind this application, push an evaluation context, and then simply unwind the stack when the value was encountered. It is possible that testBonusRating200 was encountering such a situation a very large number of times.

July 3, 2003: Initial benchmarking for the lecc machine.

The lecc machine is a graph reducer implemented in java. The lecc packager generates Java code from the compilers expression form. This code is derived from the core Java classes which implement the graph reducer. Initially the lecc machine generates one top-level class (and attendant Java source file) for each CAL module. Each supercombinator in the module generates an inner class of the module class. Also, for each supercombinator, a static singleton of the appropriate inner class is generated as a member of the module class. Data types are handled similarly to supercombinators. An inner class is generated for each data type. Then for each constructor of the data type an inner class of the data type class is generated.

	g-machine	Lecc	% change
TestBonusRating200	27991	35824	(28)%
TestBonusRating200Eta	271	2914	(975)%
BadRegressionGemTest200_200	40659	28110	31%
Foo1000000	38445	25940	33%
ForeignFunctionOverhead1000000	7041	2143	70%
PrimitiveOpOverhead1000000	6439	4813	25%
BoxManipulation	46247	Out of Memory	
BoxManipulation2	22562	28331	(25)%
BoxManipulation3	11717	14571	(25)%
Prime3000	74417	> 5 minutes. Failed to terminate.	
GetNthPrime5000	57032	43622	24%
M2.mainM2	170	3095	(1720)%

One thing that can be inferred from the benchmarking results is that the lecc has different performance characteristics than the g-machine. However, some tentative conclusions can be drawn.

1) The two worst cases are testBonusRating200Eta and M2.mainM2. Both of these run in under one second. Experimentation has shown that the lecc has a relatively flat startup time of between 2.5 and 3.5 seconds. Occasionally this startup time is significantly greater. The current theory is that this is a class loading issue. If, however, this startup time is factored out the results for these two benchmarks are more reasonable. Experiments with class loading will need to be performed to see if this startup time can be reduced.
2) Given that boxManipulation runs out of memory in lecc it is obvious that the g-machine is more parsimonious with memory than lecc. What needs to be determined is if this is because the lecc does not properly implement updating/laziness or if the lecc simply has more overhead than the g-machine.

3) Of the benchmarks the lecc runs to completion (ignoring the two benchmarks discussed in point 1) the worst performance relative to the g-machine is in the benchmarks that involve heavy manipulation of data structures. This is obviously an area where performance optimizations will need to be concentrated.
4) prime3000 continues to be the benchmark which behaves in a contrary fashion. Further investigation will be needed to determine why it doesn't terminate.
5) For the lecc foreignFunctionOverhead1000000 is actually faster than primitiveOpOverhead1000000. This would indicate that the implementation of primitive operations has room for improvement. At the very least they could be implemented as foreign functions.
6) The benchmarks that are faster in lecc are about 25-30% faster then the g-machine. This is a long ways from the initial figures of 75-80% faster than earlier results had indicated were possible, but should not be taken as a completely negative indicator. There have been no optimizations done to the lecc at this point and hopefully we can get some significant improvements in short order.

Areas to investigate for performance optimizations:

1) Primitive operation implementation. This is a good area to optimize as any gains here benefit all benchmarks and programs, not just specific cases.
2) Mini-interpreter/tail call optimizations. It is possible that performance gains can be realized by cutting down on the size of the java call stack used when running a supercombinator by packaging intermediate calls into an application graph which is returned to the caller to be further reduced. Another area of possible performance gains is to avoid building the application graph for a strictly evaluated fully saturated function application. Unfortunately these two techniques are at cross-purposes so further investigation into which to use (or both) is necessary.
3) Indirection chains of results. Currently indirection chains are avoided by having back pointers in the result nodes. However, this adds overhead to creating and managing these nodes and doesn't eliminate all cases of indirection chains forming.
4) Differentiating between let and letrec local variables. Currently all let variables are treated as letrec variables and accessed through an indirection node. In a lot of cases this is not necessary and is adding needless overhead.
5) Case expressions are always accessed through an indirection node allocated specifically for that purpose. However, in many cases the case expression is simply an existing variable or parameter. In these cases the indirection node is needless overhead.
6) The evaluation/unwind logic can be tightened up to deal with indirections and over saturation better. Currently the application graph is walked each time the supercombinator or argument count is required. There may be ways to cache this information so that it isn't re-calculated as often.
7) Currently when a foreign function is applied a new RTSupercombinator object is created for each access instance. However, strictly speaking this is only required for foreign functions with an arity of zero.
8) Currently RTCons and all derived classes are carrying around the members/logic for back walking result chains and setting indirections. This is something else that is not strictly necessary.

July 11, 2003: Changes to accessing static members in lecc.

The pattern used by the lecc machine is to create one top-level class per module. An inner class then represents each supercombinator. The module class has a static member of type RTSupercombinator for each supercombinator class and a static instance of each supercombinator class is created. Code then refers to a supercombinator by referencing the static member of the appropriate module class.

In the baseline benchmarks for the lecc a noted discrepancy was that the foreign function benchmark ran faster than the primitive op benchmark. This was surprising as the benchmarks are identical, but for the fact that one uses a foreign function for subtraction, while the other uses a primitive operation. Investigation showed that the difference was due to the foreign function allocating a new instance of the foreign function supercombinator class for each iteration, while the primitive op benchmark referenced a static member (subtractInt) in the Prelude class.
After some experimentation it was determined that the most effective solution was to have the supercombinator class initiate a local reference to refer to the static member in the Prelude class and use that local reference thereafter.

	g-machine	Lecc	Lecc2	% improvement lecc2 vs. lecc	% improvement lecc2 vs. g
PrimitveOpOverhead1000000	7303	6020	3395	44%	54%
ForeignFunctionOverhead1000000	9780	3156	3322	(5)%	66%
TestBonusRatings200	37213	46746	38451	18%	(3.3)%
BadRegressionGemTest200_200	51073	39436	33867	14%	34%
GetNthPrime5000	48623	54346	50372	7.3%	(3.5)%
Foo1000000	59095	36418	31872	12.5%	46%
BoxManipulation3	14607	20965	15883	24%	(8.7)%

All of the benchmarks showed a reasonable improvement with the change, except for the foreign function overhead benchmark. This is not too surprising, as before the change the foreign function benchmark involved no access to static members. The difference between the lecc and the g-machine is also narrowing for those cases where the g-machine is still faster.

July 16, 2003: Changes to organization of lecc generated code.

Changed the code generation in lecc to create a java package for each CAL module. A top-level class in the appropriate package then represents each supercombinator. Each supercombinator class has a singleton instance accessed through a static factory method. Data types are still represented by top-level classes containing one inner class for each constructor.

	g-machine	Lecc	Lecc2	% improvement lecc2 vs. lecc	% improvement lecc2 vs. g
PrimitiveOpOverhead1000000	6389	5035	2349	53%	63%
ForeignFunctionOverhead1000000	7761	2739	2341	14.5%	70%
TestBonusRating200	30403	34617	27447	20%	10%
BadRegressionGemTest200_200	41720	30309	23136	24%	45%
GetNthPrime5000	58144	44061	34412	22%	41%
Foo1000000	39278	30230	21138	30%	46%
BoxManipulation2	20399	11582	7000	40%	66%
Boxmanipulation3	11937	15845	11074	30%	7%

Two things should be noted here:

1) This pattern of code generation gives better performance than both the original pattern in lecc1 and the revised pattern documented and benchmarked on July 11th.
2) For all the benchmarks run in this instance the lecc generated code runs faster than the interpreted g-machine (though in some cases by a fairly narrow margin).

July 17, 2003: Use of java switches for CAL case statements and optimization of top-level conditional expressions in LECC.

Modified the code generate so that an actual java switch is used to represent CAL case statements. This is possible because the case lifting in the compiler ensures that all CAL case statements are at the top level of the containing supercombinator and that nested cases don't exist. This requires being able to interpret the tag in the CAL case statement as an integer (since java switches only on integers). This was accomplished by overriding RTValue.getIntegerValue() in the CAL_Boolean class to return 1 for true and 0 for false. Similarly the getIntegerValue() function is overridden in classes derived from RTCons to deliver the ordinal value corresponding to the specific data constructor.
With the use of a java switch the let variables for each alternate in the CAL case statement could be moved into the java switch alternate. This avoids the overhead, in this situation, of building the graph for let variables that will never be used.
Moving the let variables into the switch alternates removes the need for the switch expression to be assigned to a previously allocated RTIndirection node. This avoids the extra allocation of the node and also makes accessing the switch expression value more direct.

Conditional expressions encountered in a top-level code generation context were also optimized. In this case the code generated now uses the java 'if-then-else' rather than the '?' operator. For the purpose of optimizing conditional expressions a top-level context is either: 1) the top level of a supercombinator, 2) the top level of a switch alternate.
As with the use of the switch above changing to using 'if-then-else' allows the let variables for the two branches of the conditional be moved into the branches of the 'if-then-else' avoiding the overhead of building the graph for a let variable that will not be used.

	g-machine	Lecc: pre-case	Lecc: post-case	% improvement lecc pre vs. lecc post	% improvement lecc vs. g
PrimitiveOpOverhead1000000	6389	2349	2336	0.5%	63%
ForeignFunctionOverhead1000000	7761	2341	2321	1%	70%
TestBonusRating200	30403	27447	25642	7%	16%
BadRegressionGemTest200_200	41720	23136	22049	5%	47%
GetNthPrime5000	58144	34412	32469	6%	44%
Foo1000000	39278	21138	19936	6%	49%
BoxManipulation2	20399	7000	6645	5%	67%
BoxManipulation3	11937	11074	10979	1%	8%

There are no huge gains in performance here, although a gain of 5-7% is large enough to be considered 'real'. In the case of the primitive op and foreign function benchmarks there was no expectation of improvement, as these benchmarks don't involve case statements. In the case of BoxManipulation3 any gains from the optimizations are probably overshadowed by the overhead of modifying all the allocated data structures. This is obviously an area that will bear future consideration.
Looking forward there are some further code generation enhancements that can be built on top of these. First the use of lazy extractors in case statements can be optimized, and possibly eliminated. Second, when a conditional is encountered at a top-level each of its branches could be considered the start of a new top-level context. This would allow for the optimization of nested conditionals into 'if-then-else' form.

July 22, 2003: Optimization of 'and' and 'or'.

Uses of Boolean and/or were previously always being generated as a function application in two arguments. Since the java operations && and || are naturally lazy in their second argument (i.e. the second argument is only evaluated if necessary) an optimization to implement and/or as primitive operations was introduced. This optimization permitted the use of java && and || when generating code for a strict context.
An additional change was to remove the use of lazy extractor functions when accessing the members of a data type.

	g-machine	Lecc: pre-optimization	Lecc: post-optimization	% improvement lecc pre vs. lecc post	% improvement lecc vs. g
PrimitiveOpOverhead1000000	6389	2336	2386	(2%)	63%
ForeignFunctionOverhead1000000	7761	2321	2369	(2%)	69%
TestBonusRating200	30403	25642	18342	28%	40%
BadRegressionGemTest200_200	41720	22049	16551	25%	60%
GetNthPrime5000	58144	32469	23229	28%	60%
Foo1000000	39278	19936	16466	17%	58%
BoxManipulation	46247	------	24611	-----	47%
BoxManipulation2	20399	6645	6282	5%	69%
BoxManipulation3	11937	10979	6339	42%	47%

A few things to note:

1) The primitive-op overhead and foreign function benchmarks showed a slightly negative change. However, given the short duration of the benchmarks and the small size of the change this is not considered significant.
2) The boxManipulation benchmark is now running in the lecc2 machine. Previously this benchmark failed to terminate due to an out-of-memory error.

August 15, 2003: Optimizations to boxing/un-boxing of values and to let variables.

In cases where a foreign function or primitive-op is an argument to another foreign function or primitive-op the result of the argument is no longer boxed (i.e. put into a cal node object). This only applies where the code is

being generated in a strict context. Similarly the result of a conditional expression is no longer boxed into a cal Boolean object.
Let variables that aren't referred to by other let variables are no longer put into an indirection. Also, a let variable whose definition is a supercombinator (caused by lambda lifting) or a literal is no longer declared as the code generator fixes up references to the let variable to refer to the literal/supercombinator.

	g-machine	Lecc: pre-optimization	Lecc: post-optimization	% improvement lecc pre vs. lecc post	% improvement lecc vs. g
PrimitiveOpOverhead1000000	6389	2340	2203	6%	65%
ForeignFunctionOverhead1000000	7761	2313	2246	3%	71%
TestBonusRating200	30403	18393	16541	10%	46%
TestBonusRating200Eta	13	531	484	9%	(36%)
BadRegressionGemTest200_200	41720	16220	18133	(12%)	57%
GetNthPrime5000	58144	22873	22469	2%	61%
Foo1000000	39278	16137	16010	1%	59%
BoxManipulation	46247	24182	23928	1%	48%
BoxManipulation2	20399	6155	6008	2%	70%
BoxManipulation3	11937	6179	6112	1%	49%
PrimitiveOpOverhead2_1000000	7978	2584	2203	15%	72%
LetTest1000000	21828	9417	8305	12%	62%

Two new benchmarks have been added to the performance benchmark suite. The descriptions are in the benchmark section but primitiveOpOverhead2_1000000 involves repeated evaluations of chained primitive operations while letTest1000000 involves recursive evaluation of a function that declares/evaluates let variables.

Some things to note in these results:

1) For most of the benchmarks the change was minimal, i.e. 1-2%.
2) The original primitive-op and foreign function benchmarks showed a slightly better improvement, probably due to the optimization to avoid boxing conditional expression results.
3) The two new benchmarks, designed to better test the areas being optimized, both show a good improvement of 12-15%.
4) Two somewhat anomalous results stand out. The more minor of the two is the 10% improvement in testBonusRating200. The more serious is the 12% degradation of badRegressionGemTest200_200. This performance degradation is puzzling and will bear further investigation.

A further optimization to the usage of let variables that are defined as a literal would allow direct use of the literal when the let variable is referenced as an argument to a string primitive op. Currently the let variable reference is resolved to a reference to a cal value object. While this would probably give a nice further improvement to the let test benchmark the situation where a let variable is simply assigned a literal value isn't that common so this optimization isn't really justified at this point in time.

August 15, 2003: Changes to prelude to use constrained instances.

	g-machine (old prelude)	g-machine (new prelude)	% change	Lecc (old prelude)	Lecc (new prelude)	% change
PrimitiveOpOverhead1000000	6389	6369	0.3%	2203	2213	(0.4%)
ForeignFunctionOverhead1000000	7761	8135	(5%)	2246	2243	0.1
TestBonusRating200	30403	30757	(1%)	16541	16517	0.1
BadRegressionGemTest200_200	41720	41542	0.4%	18133	18186	(0.3%)
GetNthPrime5000	58144	58106	0%	22469	22492	(0.1%)
Foo1000000	39278	49160	(25%)	16010	16033	(0.1%)
BoxManipulation	46247	48263	(4%)	23928	23945	0%
BoxManipulation2	20399	21458	(5%)	6008	6019	(0.1%)
BoxManipulation3	11937	11553	3%	6112	6136	(0.4%)
PrimitiveOpOverhead2_1000000	7978	8038	(1%)	2203	2204	0%
LetTest1000000	21828	21788	0%	8305	8232	0.9%

Generally speaking performance changes with constrained instances were negligible in the lecc machine. However the g-machine didn't fare as well. Three benchmarks suffered performance degradation in the 4-5% range. The real aberration is the foo100000 benchmark. On the g-machine this benchmark went down by 25%. Bo is going to look into this.

September 11, 2003: Integration of lecc into ICE.

	Tester - execute	Tester - unwind	ICE - execute	ICE - unwind	% change execute	% change unwind
PrimitiveOpOverhead1000000	2223	0	3775	0	(70%)	-
ForeignFunctionOverhead1000000	2253	0	3849	0	(70%)	-
TestBonusRating200	160	16026	427	15859	(162%)	1%
TestBonusRating200Eta	150	344	424	824	(183%)	(145%)
BadRegressionGemTest200_200	18366	0	18256	0	0.6%	-
GetNthPrime5000	22729	0	23283	0	(2.4%)	-
Foo1000000	20516	0	23357	0	(13.8%)	-
BoxManipulation	23000	0	21267	0	7.5%	-
BoxManipulation2	27	5938	196	7353	(625%)	(24%)
BoxManipulation3	5905	0	4459	0	24.5%	-
PrimitiveOpOverhead2_1000000	2220	0	3842	0	(73%)	-
LetTest1000000	8258	0	9507	0	(15%)	-

The initial integration of lecc into ICE has caused some performance degradation. It was expected that unwinding would be slower in ICE as the whole client side unwind mechanism is introduced into the performance timings. However, execution also is slower in almost all cases. One possibility is that the use of the URLClassLoader is slowing down performance when compared to using the simple test application.

September 18, 2003: Client vs. Server JVM performance for G-Machine and LECC in ICE

	G-Machine			LECC2			LECC2 vs. G % diff.	
	client	server	% diff.	client	server	%diff	client	server
TestBonusRating200	31175	23154	25.73%	18406	14511	21.16%	40.96%	37.33%
TestBonusRating200Eta	301	200	33.55%	281	231	17.79%	6.64%	-15.50%
BadRegressionGemTest200_200	46797	25878	44.70%	20349	16995	16.48%	56.52%	34.33%
GetNthPrime5000	62941	35160	44.14%	28811	22743	21.06%	54.23%	35.32%
Foo1000000	59365	26999	54.52%	24005	18506	22.91%	59.56%	31.46%
ForeignFunctionOverhead1000000	8312	4237	49.03%	2353	1943	17.42%	71.69%	54.14%
PrimitiveOpOverhead1000000	6560	3455	47.33%	2324	1803	22.42%	64.57%	47.81%
BoxManipulation	51544	33468	35.07%	26118	21524	17.59%	49.33%	35.69%
BoxManipulation2	22783	20629	9.45%	9924	7961	19.78%	56.44%	61.41%
BoxManipulation3	12728	7221	43.27%	4997	4797	4.00%	60.74%	33.57%
PrimitiveOpOverhead2_1000000	8021	4376	45.44%	5708	1863	67.36%	28.84%	57.43%
LetTest1000000	21521	12128	43.65%	8382	6099	27.24%	61.05%	49.71%

It is interesting to note that the LECC machine doesn't experience as large a performance gain as the G-machine when going from the client jvm to the server jvm. As a result LECC vs. G-machine performance is not as good in the server jvm scenario. Why the LECC doesn't see as much benefit from the server jvm is somewhat open to conjecture. The G-machine has a much smaller class footprint, and once the different node classes are loaded no more class loading is necessary. It is possible that the server jvm handles this scenario better than the scenario in the LECC machine where each supercombinator is a different class, thereby requiring a lot more class loading.

September 25, 2003: Optimizations to let variable generation.

In cases where a let variable is defined as: 1) A literal, 2) One of the supercombinators formal arguments, or 3) a previously defined let variable the definition of the let variable can be optimized out in the code generation by directly substituting the definition of the let variable for every reference to it.

	Lecc	Lecc2	% change
LetTest1000000	10011	10075	(1%)

Somewhat surprisingly the optimizations produced a small decrease in performance. Trying various combinations of changes produced the following results.

	Remove optimization of lets defined as a literal.	Make literals into method local variables.	Make literals into non-static class members.	Add back optimization of lets defined as a literal.	Remove calls to evaluate on a literal.
LetTest1000000	9844	10165	9771	9770	9644

The performance degradation in the first table would appear to be due to the optimization that substitutes direct referrals to the literal object for referrals to a let variable defined as the literal object. This is shown by the improvement in performance when that particular optimization is removed. Given that the optimization

removed the creation of local variables the most likely cause of the problem is accessing static class members. Changing the literals to be method local variables created an even worse performance hit. However, changing the literals to non-static class members showed no performance hit. Adding back in the optimizations to remove local variables defined as literals and then the further optimization this allows of removing calls to evaluate on a literal brought performance to the point of being 3.7% better than in lecc.
Moving the literal variables from static class members to non-static class members shouldn't pose a problem in terms of greater memory usage as there is only a single instance allowed for each supercombinator class.

September 30, 2003: Optimization of perfectly saturated applications in LECC.

This optimization addresses the case of an application chain that applies a supercombinator to the exact number of required arguments. When this situation is identified a specialized graph node is created which contains the supercombinator and the arguments in a single node. This reduces the amount of graph building that takes place in performing the chain of applications. This optimization applies for supercombinators with an arity of up to four.
In addition the code generation was enhanced. In cases where a supercombinator has an arity of one to four a secondary body function is generated in the corresponding java class. The function (fN where N is the arity) can be called directly by the specialized application nodes. This avoids having to build and unwind the graph of the arguments.
As can be seen in the results table this was a good optimization. The smallest improvement was 17.47%, on the boxManipulation3 benchmark. The tightly recursive benchmarks were a good fit for this optimization and received gains of up to 63%.

	Before Optimizing	After Optimizing	% Improvement
TestBonusRating200	16634	13266	20.25%
TestBonusRating200Eta	184	121	34.24%
BadRegressionGemTest200_200	20640	15866	23.13%
GetNthPrime5000	27937	21577	22.77%
Foo1000000	23517	18770	20.19%
ForeignFunctionOverhead1000000	2330	1012	56.57%
PrimitiveOpOverhead1000000	2293	975	57.48%
BoxManipulation	28324	22469	20.67%
BoxManipulation2	8028	6727	16.21%
BoxManipulation3	6573	5425	17.47%
LetTest1000000	8185	6002	26.67%
PrimitiveOpOverhead2_1000000	2283	964	57.77%
ApplyTest	11240	4049	63.98%

Given the success of this optimization further exploration in this area is warranted. Currently the optimization only occurs when there is an application chain with exactly the right number of arguments. Observation has shown that the CAL overloading mechanism frequently leads to the generation of over-saturated application chains. Potentially the optimization could be expanded to include this case.
Another potential extension of this optimization is in dealing with perfect applications in strict execution contexts. In this case we are currently creating one of the specialized application nodes and then calling 'evaluate()' on it. We should be able to avoid the creation of the application node and instead generate a direct call to the 'fn()' function of the supercombinator class.

October 3, 2003: Optimizations to fully saturated applications in a strict context.

As an extension of the previous optimization, which introduced special graph nodes for fully saturated applications of up to 4 arguments, fully saturated applications in a strict context were optimized. In this situation a graph node to represent the application was being created and then having its 'evaluate()' function called immediately. This resulted in the 'fn()' member of the supercombinator being called with the cached arguments and then the resulting graph would be unwound. Instead the creation of the initial application node can be skipped and the generated code can substitute a direct call to the appropriate 'fn()' member.

	Before Optimizing	After Optimizing	% Improvement
TestBonusRating200	13266	12915	2.65%
TestBonusRating200Eta	121	103	14.88%
BadRegressionGemTest200_200	15866	14067	11.34%
GetNthPrime5000	21577	21166	1.90%
Foo1000000	18770	18223	2.91%
ForeignFunctionOverhead1000000	1012	998	1.38%
PrimitiveOpOverhead1000000	975	971	0.41%
BoxManipulation	22469	22459	0.04%
BoxManipulation2	6727	6640	1.29%
BoxManipulation3	5425	4877	10.10%
LetTest1000000	6002	5856	2.43%
PrimitiveOpOverhead2_1000000	964	955	0.93%
ApplyTest	4049	3872	4.37%

The two big winners with this manipulation were the regression gem and box manipulation 3 benchmarks. The TestBonusRating200Eta benchmark also showed significant improvement but is such a fast/transient benchmark that too much shouldn't be read into this.

October 17, 2003: Optimizations to switches.

Optimizations to various special case switches were experimented with. These included:

1) Transforming a switch on a Boolean to an if-then-else.
2) Changing single case switches to an if-then-else.
3) Changing cases with two alternates, one of which was a default, to an if-then-else.
4) Ordering the cases in a switch and filling in any missing integer values to create a contiguous block of cases.

Experimentation showed no significant differences in performance with the various changes. Optimization 1 was left intact as it generates more readable source code. Also, optimization 4 was left intact, as some compilers/runtimes are supposedly more optimal in this situation.

	Before Optimizing	Std Deviation	After Optimizing	Std Deviation	% Improvement
TestBonusRating200	11486	1.07%	11553	0.90%	-0.58%
BadRegressionGemTest200_200	12561	0.04%	12922	0.14%	-2.87%

GetNthPrime5000	17188	0.58%	17037	0.82%	0.88%
Foo1000000	15647	0.13%	15679	0.03%	-0.20%
ForeignFunctionOverhead1000000	9644	0.09%	9594	0.09%	0.52%
PrimitiveOpOverhead1000000	9524	0.01%	9304	0.01%	2.31%
BoxManipulation	18704	0.85%	18540	1.71%	0.88%
BoxManipulation2	6175	10.51%	5808	0.62%	5.94%
LetTest1000000	5949	0.01%	5976	0.08%	-0.45%
PrimitiveOpOverhead2_1000000	9390	0.14%	9093	0.00%	3.16%
ApplyTest	3886	0.01%	3856	0.21%	0.77%

October 17, 2003: Type checking when unwinding over saturated applications.

If the compiler generates an application of a non-function to a set of arguments the runtime will go into an infinite loop. The fix for this was to check the actual type of the result value from evaluating the left hand side of the over saturated graph. There was some concern that adding this run time type identification into the unwind loop would be a performance hit. Subsequent testing showed that there was no significant performance change.

	Before Optimizing	**Std Deviation**	**After Optimizing**	**Std Deviation**	**% Improvement**
TestBonusRating200	12014	0.57%	11987	0.82%	0.22%
BadRegressionGemTest200_200	13413	0.25%	13539	1.63%	-0.94%
GetNthPrime5000	17508	0.31%	17816	0.74%	-1.76%
Foo1000000	16317	0.13%	16233	0.10%	0.51%
ForeignFunctionOverhead1000000	9601	0.13%	9634	0.22%	-0.34%
PrimitiveOpOverhead1000000	9350	0.14%	9337	0.18%	0.14%
BoxManipulation	19638	0.90%	19624	0.89%	0.07%
BoxManipulation2	6035	0.33%	6392	9.06%	-5.92%
LetTest1000000	6092	0.08%	6112	0.08%	-0.33%
PrimitiveOpOverhead2_1000000	9373	0.01%	9333	0.00%	0.43%
ApplyTest	4099	0.12%	4039	0.12%	1.46%

October 22, 2003: Optimizations to use of supercombinator factory methods.

Previously the body of a supercombinator would get an instance of any referenced supercombinator by calling the referenced supercombinators factory method (make). However, if the referenced supercombinator is safe (i.e. no chance of a side-effect) the factory method simply is returning a singleton instance. Because the factory method is a static method there is extra overhead in calling it.

For referenced supercombinator's that are safe a non-static class member can be created which is a local reference to the singleton instance. This class member will be initialized once, when the class instance is created. The body of the supercombinator can then use this local reference rather than call the referenced supercombinator's factory method.

	Before Optimizing	**After Optimizing**	**Std Deviation**	**% Improvement**
TestBonusRating200	14887	14046	0.64%	5.50 %
BadRegressionGemTest200_200	17729	16690	1.50%	5.80%

GetNthPrime5000	22853	21644	0.38%	5.30%
Foo1000000	21521	20426	0.40%	5.10%
ForeignFunctionOverhead1000000	12671	11820	0.28%	6.70%
PrimitiveOpOverhead1000000	12764	11677	0.06%	8.50%
BoxManipulation	23801	22118	1.26%	7.10%
BoxManipulation2				
LetTest1000000	7681	7487	0.32%	2.50%
PrimitiveOpOverhead2_1000000	11463	10872	0.78%	5.10%
ApplyTest	5468	4980	2.08%	8.90%

In the course of benchmarking this change a strange new behaviour was seen in the BoxManipulation2 benchmark. Every other run (i.e. runs # 2, 4, 6, etc.) took significantly longer then previous tests. So while odd numbered runs took approximately 7 seconds (in-line with previous results) the even numbered runs took approximately 45 seconds. This behaviour will be investigated further.
Update: Subsequent investigation showed that the strange behaviour of boxManipulation2 could be removed by increasing the heap size available to the jvm via the –Xms and –Xmx command line options. This would indicate that it is a memory use/garbage collection issue. Probably the result of a run, which is held by a static member of the supercombinator class is slow to be garbage collected after deleting the class loader instance that loaded the class.

October 29, 2003: Changes to performance timing in lecc.

The timing mechanism in the lecc runtime was adjusted to include the loading/resolving of the initial target class in the timing. The timing in the g-machine includes the 'loading' of the target supercombinator and so should the timing in lecc, especially since in the case of the lecc machine the loading of the initial target class will often include loading/resolving referenced classes and can be quite significant. This overhead is most significant on the initial run of a benchmark, despite the fact that all classes are being unloaded between each run. This would indicate that after the initial run the class files are probably in a disk cache, thus speeding up subsequent loading of the .class files.

	Before Change	First Run	After Change	First Run	Change	Change First Run
TestBonusRating200	11199	11237	11422	13059	-1.99%	-16.21%
BadRegressionGemTest200_200	12101	12028	12468	12729	-3.03%	-5.83%
GetNthPrime5000	16304	16363	16414	17495	-0.67%	-6.92%
Foo1000000	15315	15472	15356	15193	-0.27%	1.80%
ForeignFunctionOverhead1000000	8609	8573	8612	8673	-0.03%	-1.17%
PrimitiveOpOverhead1000000	8449	8443	8469	8483	-0.24%	-0.47%
BoxManipulation	10916	11046	11103	11267	-1.71%	-2.00%
BoxManipulation2	5314	5267	5484	5388	-3.20%	-2.30%
PrimitiveOpOverhead2_1000000	8512	8873	8546	8623	-0.40%	2.82%
LetTest1000000	6052	6019	6132	6409	-1.32%	-6.48%
ApplyTest	3605	3626	3599	3666	0.17%	-1.10%
M2.mainM2	367	1342	7678	17445	-1992.10%	-1199.93%

November 4, 2003: Changes to Class Loading

The CALClassLoader and CAL_ModuleClassLoader were re-written to conform more closely to generally accepted practice in writing custom class loaders. Primarily this consisted of overriding the findClass() method, rather than the loadClass() method and making sure not to break/circumvent the class loading delegation model. The results were, generally speaking, positive/neutral. The most significant decline was primitiveOpOverhead2_10000000, which only went down by 1.7%.

	Before Optimizing (ms)	After Optimizing	% Improvement
TestBonusRating200	11503	11079	3.69%
BadRegressionGemTest200_200	12578	12157	3.35%
GetNthPrime5000	16597	16377	1.33%
Foo1000000	15716	15309	2.59%
ForeignFunctionOverhead1000000	8759	8702	0.65%
PrimitiveOpOverhead1000000	8532	8535	-0.04%
BoxManipulation	11086	10885	1.81%
BoxManipulation2	5421	5380	0.76%
LetTest1000000	6115	6052	1.03%
PrimitiveOpOverhead2_1000000	8589	8735	-1.70%
ApplyTest	3688	3631	1.55%

The implementation of the CALClassLoader includes a map (implemented via HashMap) of module class loaders where the key is the name of the package corresponding to the module. When CALClassLoader.findClass() is called the final package name has to be parsed out of the fully qualified class name. This involves finding the index of the last two appearances of '.' and taking the text between as a sub string. As an experiment the module class loaders were put into the map using the fully qualified package name as a key. This was done on the assumption that finding the index of the '.' character was an expensive operation. The results show that creating/hashing a larger sub string is actually slightly more expensive than an additional call to 'lastIndexOf()' and creating/hashing a shorter sub string.

	Before Optimizing (ms)	After Optimizing	% Improvement
TestBonusRating200	11079	11146	-0.60%
BadRegressionGemTest200_200	12157	12291	-1.10%
GetNthPrime5000	16377	16330	0.29%
Foo1000000	15309	15372	-0.41%
ForeignFunctionOverhead1000000	8702	8796	-1.08%
PrimitiveOpOverhead1000000	8535	8742	-2.43%
BoxManipulation	10885	11072	-1.72%
BoxManipulation2	5380	5453	-1.36%
LetTest1000000	6052	6042	0.17%
PrimitiveOpOverhead2_1000000	8735	8740	-0.06%
ApplyTest	3631	3658	-0.74%

As another experiment the compilation process was modified to include pre-loading all the generated classes in an attempt to remove the overhead of class loading from the execution times. Interestingly this change actually slowed down all the benchmarks except M2.mainM2. For most of the benchmarks the working set of classes required to run the benchmark is relatively small. It would seem that having all the generated classes loaded creates a large enough memory footprint that paging of the virtual memory system occurs when accessing the

classes in the working set. For M2.mainM2, where class loading is such an overriding factor there is a remarkable improvement. Not only is the average run time significantly faster, but also the first run is much more consistent with the subsequent runs. Given that M2.mainM2 is not typical in nature of most programs pre-loading all classes would appear to be counterproductive at this time.

	Before Change	First Run	After Change	First Run	% Change	% Change First Run
TestBonusRating200	8529	9534	9126	9243	-7.00%	3.05%
BadRegressionGemTest200_200	12264	12458	13142	13240	-7.16%	-6.28%
GetNthPrime5000	16590	16805	17279	17286	-4.15%	-2.86%
Foo1000000	15218	15242	15986	16023	-5.05%	-5.12%
ForeignFunctionOverhead1000000	8642	8763	8749	8763	-1.24%	0.00%
PrimitiveOpOverhead1000000	8509	8413	8679	8713	-2.00%	-3.57%
BoxManipulation	11149	11217	11547	11727	-3.57%	-4.55%
BoxManipulation2	5333	5308	5932	6309	-11.23%	-18.86%
PrimitiveOpOverhead2_1000000	8505	8823	8659	8743	-1.81%	0.91%
LetTest1000000	6065	6109	6259	6299	-3.20%	-3.11%
ApplyTest	3645	3675	3722	3795	-2.11%	-3.27%
M2.mainM2	3405	13379	1355	2274	60.21%	83.00%

As a further experiment in pre-loading classes just the class from the Prelude module were pre-loaded. The idea was that these were classes more likely to be used by most programs and that a pre-loading a subset of all classes might not cause the memory paging issues seen when pre-loading all classes.
Unfortunately the results weren't significantly better. In fact M2.mainM2 slowed down significantly from the pre-load all classes scenario and the other benchmarks were all still slower on average then with no pre-loading. It is interesting to note that some of the benchmarks do show a slight improvement in performance on the first run. Going forward it may be determined that optimizing for first run performance is important to enhance real world user experience. If this occurs it will probably be worth re-examining the concept of pre-loading a small core set of classes as part of the compilation process.

Pre-Load Prelude classes

	Before Change	First Run	After Change	First Run	% Change	% Change First Run
TestBonusRating200	8529	9534	8633	8933	-1.22%	6.30%
BadRegressionGemTest200_200	12264	12458	12638	12858	-3.05%	-3.21%
GetNthPrime5000	16590	16805	16658	16665	-0.41%	0.83%
Foo1000000	15218	15242	15553	15082	-2.20%	1.05%
ForeignFunctionOverhead1000000	8642	8763	8695	8713	-0.61%	0.57%
PrimitiveOpOverhead1000000	8509	8413	8515	8513	-0.07%	-1.19%
BoxManipulation	11149	11217	11159	11227	-0.09%	-0.09%
BoxManipulation2	5333	5308	5587	5688	-4.76%	-7.16%
PrimitiveOpOverhead2_1000000	8505	8823	8583	8672	-0.92%	1.71%
LetTest1000000	6065	6109	6099	6089	-0.56%	0.33%
ApplyTest	3645	3675	3642	3656	0.08%	0.52%
M2.mainM2	3405	13379	5221	8112	-53.33%	39.37%

November 4, 2003: Optimization of supercombinator references in fully saturated strict applications.

In cases where a fully saturated application chain occurred in a strict context the generated code was optimized to directly call the supercombinators 'fn' body function, rather than create an application graph. However the generated code was still using the factory method to obtain an instance of the supercombinator (ex. Foo.make().f3(a, b, c)). The previous optimization to supercombinator references can be applied here, creating a class field to hold an instance of Foo and simply reference this in the function body.

	Before Optimizing (ms)	After Optimizing	% Improvement
TestBonusRating200	8739	8722	0.19%
BadRegressionGemTest200_200	12565	12468	0.77%
GetNthPrime5000	16835	16794	0.24%
Foo1000000	15165	15220	-0.36%
ForeignFunctionOverhead1000000	8729	8773	-0.50%
PrimitiveOpOverhead1000000	8565	8546	0.22%
BoxManipulation	11280	11073	1.84%
BoxManipulation2	5929	5414	8.69%
PrimitiveOpOverhead2_1000000	8545	8586	-0.48%
LetTest1000000	6219	6115	1.67%
ApplyTest	3669	3668	0.03%

Generally speaking changes in performance were too small to be significant, except for boxManipulation2. This is somewhat surprising as boxManipulation2 exercises the same functionality as boxManipulation with the difference of unwinding the resulting container tree, rather than collapsing it to a single number.

November 4, 2003: Optimization of application chains with more than 4 arguments.

A previous optimization used special purpose graph nodes to replace application graphs for fully saturated supercombinators of up to 4 arguments. This optimization was extended for supercombinators of up to 8 arguments.

	Before Change	After Change	% Improvement
TestBonusRating200	11403	8529	25.20%
BadRegressionGemTest200_200	12414	12264	1.21%
GetNthPrime5000	16567	16590	-0.14%
Foo1000000	15335	15218	0.76%
ForeignFunctionOverhead1000000	8852	8642	2.37%
PrimitiveOpOverhead1000000	8742	8509	2.67%
BoxManipulation	11122	11149	-0.24%
BoxManipulation2	5324	5333	-0.17%
PrimitiveOpOverhead2_1000000	8756	8505	2.87%
LetTest1000000	6025	6065	-0.66%
ApplyTest	3678	3645	0.90%

Since most of the benchmarks don't make use of higher arity functions no change was expected and the actual differences were insignificant. However, for TestBonusRating200, which does use some higher arity functions, the improvement was significant.

A more general application of this technique, which could be applied to arbitrary length application chains, would seem to be desirable and bears further thought.

November 12, 2003: Consolidation of the body functions in supercombinators.

Currently when the java class corresponding to a CAL supercombinator is generated it contains two versions of the method implementing the supercombinator logic: the 'f' method and the 'fn' method (where n is the arity of the supercombinator).
The 'f' method takes the root of the application chain containing the arguments to the supercombinator. The 'fn' method takes each of the supercombinator arguments as a method argument. Except for the code the 'f' function needs to 'unpack' the arguments from the application chain the bodies of the two methods are identical. This leads to the question of consolidation: the 'f' method can simply unpack the arguments from the application chain and then call the 'fn' method.

Consolidate f and fn methods- Client JVM

	Before Consolidation		After Consolidation			
	First Run	**Average**	**First Run**	**Average**	**% Improvement 1st run**	**% Improvement**
TestBonusRating200	9454	8568	9664	8502	-2.22%	0.77%
BadRegressionGemTest200_200	12418	12358	13019	12291	-4.84%	0.54%
GetNthPrime5000	17035	16333	17035	16364	0.00%	-0.19%
Foo1000000	15503	15311	15754	15362	-1.62%	-0.33%
ForeignFunctionOverhead1000000	8632	8592	8653	8683	-0.24%	-1.06%
PrimitiveOpOverhead1000000	8483	8439	8462	8452	0.25%	-0.15%
BoxManipulation	11316	11066	11607	11083	-2.57%	-0.15%
BoxManipulation2	5468	5527	5848	5658	-6.95%	-2.37%
PrimitiveOpOverhead2_1000000	8562	8545	8603	8486	-0.48%	0.69%
LetTest1000000	6219	6042	6309	6065	-1.45%	-0.38%
ApplyTest	3676	3592	3685	3602	-0.24%	-0.28%
MainM2	15101	2483	15874	2470	-5.12%	0.52%

consolidate f and fn server

	Before Consolidation		After Consolidation			
	First Run	**Average**	**First Run**	**Average**	**% Improvement 1st run**	**% Improvement**
TestBonusRating200	7861	6819	8292	6917	-5.48%	-1.44%
BadRegressionGemTest200_200	10294	9961	11187	9854	-8.67%	1.07%
GetNthPrime5000	13149	13199	14121	13156	-7.39%	0.33%
Foo1000000	11596	11619	12058	11740	-3.98%	-1.04%
ForeignFunctionOverhead1000000	6910	6883	7070	6993	-2.32%	-1.60%
PrimitiveOpOverhead1000000	6700	6846	6740	7050	-0.60%	-2.98%
BoxManipulation	9193	9053	9704	8986	-5.56%	0.74%
BoxManipulation2	4817	6496	5388	6813	-11.85%	-4.88%
PrimitiveOpOverhead2_1000000	6900	6803	6831	6947	1.00%	-2.12%
LetTest1000000	4857	4629	4897	4650	-0.82%	-0.45%

ApplyTest	3065	3031	3114	3068	-1.60%	-1.22%
MainM2	10765	2403	11116	2630	-3.26%	-9.45%

The results of this change are somewhat mixed. For the client JVM eight out of twelve benchmarks show a negative change. However the change is less than one percent in six of the eight. The server JVM shows a negative change for nine out of twelve benchmarks and the changes are more significant, with only one of the nine being below one percent.
The difference in size of the generated class files on disk was 0.03 megabytes with a reduction from 2.21 to 2.18 megabytes for a disk usage reduction of 1.3%.

Based on the generally negative performance impact, more notably with the server JVM, and the small change in class file size the methods will be left unconsolidated for now.

November 14, 2003: Adding statistics generation to lecc2.

The g-machine keeps track of the number of instructions executed in the course of running a program and this statistic is occasionally useful as an absolute measure of performance. The lecc machine doesn't operate as an interpreter and therefore doesn't have the concept of instructions. However, there are some equivalent measures that can be accumulated.
For the lecc there are three counts that have meaning: 1) reductions, 2) supercombinator executions, 4) data type instances. Due to the distributed nature of the lecc there are some difficulties in generating the statistics. As a result there is a flag in the SourceGenerationConfiguration class that turns on collecting these statistics. Changing the value of this flag requires re-generating the generated java code.

Add statistics counts.

	Before Change (ms)		After Change (ms)			
	First Run	**Average**	**First Run**	**Average**	**% Improvement 1st run**	**% Improvement**
TestBonusRating200	9995	8693	10165	8722	-1.70%	-0.33%
BadRegressionGemTest200_200	12829	12618	13080	12855	-1.96%	-1.88%
GetNthPrime5000	17105	16564	17916	17045	-4.74%	-2.90%
Foo1000000	15903	15796	15743	15873	1.01%	-0.49%
ForeignFunctionOverhead1000000	8663	8575	8642	8753	0.24%	-2.08%
PrimitiveOpOverhead1000000	8432	8412	8392	8509	0.47%	-1.15%
BoxManipulation	11326	11186	11367	11136	-0.36%	0.45%
BoxManipulation2	0	0	0	0	#DIV/0!	#DIV/0!
PrimitiveOpOverhead2_1000000	8402	8466	8372	8579	0.36%	-1.33%
LetTest1000000	6069	6035	6018	6018	0.84%	0.28%
ApplyTest	3696	3648	3755	3732	-1.60%	-2.30%
MainM2	18077	3732	21011	3762	-16.23%	-0.80%

The inclusion of generating these counts does have a small negative affect on performance so the ability to exclude the generating of these counts will be left in.

November 14, 2003: Some code statistics:

The distribution by arity of the supercombinators in the existing body of CAL code:

Arity	Supercombinator Count
0	575
1	802
2	486
3	254
4	86
5	64
6	37
7	23
8	14
9	9
10	3
11	3
13	2
15	2
Total	2360

The distribution would indicate that the current optimization of fully saturated supercombinator of up to arity 8 catches 99% of the existing supercombinators.

App chain construction:
Total application chain constructions: 4849
Under saturated chain constructions: 756
Over saturated chain constructions: 798
Distribution by length of app chain constructions:

Application chain length.	Number of occurrences.
1	2221
2	1594
3	726
4	164
5	62
6	37
7	22
8	7
9	8
10	2
11	2
13	2
15	2

Identity functions:
Supercombinators that are functionally equivalent to the identity function can potentially be optimized out of generated code. There are currently 50 supercombinators where the definition consists solely of a literal, a supercombinator reference, or a formal argument.

Runtime statistics for TestBonusRating200:
TestBonusRating200 was chosen for collecting some runtime statistics because it is a benchmark that is actually based on gems that do something an end user could find useful.

Number of reductions: 7679425
Number of over saturated chains resolved: 1276192 (16% of total)
Resolution count for methods on the left hand side of over saturated chains:

Method	Number of resolutions	Chain length
Prelude.sort	800	2
Prelude.length	1	1
Prelude.greaterThanEquals	1000	3
Prelude.equals	311000	3
Prelude.lessThan	651590	3
Prelude.zip	1	2
Prelude._getPrelude_Eq_Prelude_Ord	311000	2
Prelude.head	1000	2

November 14, 2003: Interrupting execution in lecc.

Setting a flag in the Executor object can halt execution of the g-machine, however the lecc machine currently has no provision for halting during execution. Due to the distributed nature of the lecc machine it is difficult to check a status flag without involving the use of static members in some class, for example: a static flag in the Executor class. This approach is problematic in scenarios where more than one executor is running, as there is no way to halt an individual executor.
One possible approach is to have the generated code and core runtime classes check the status of the executing threads interrupted flag. This would allow a client to signal the interruption of an individual thread of execution. The interrupted thread could then halt in a controlled manner at the next check of the flag.
As an experiment such a check was put into the evaluation loop in RTResultFunction.evaluate(). The results were discouraging!

	Before Change (ms)		After Change (ms)			
	First Run	**Average**	**First Run**	**Average**	**% Improvement 1st run**	**% Improvement**
TestBonusRating200	9995	8693	13740	13733	-37.47%	-57.98%
BadRegressionGemTest200_200	12829	12618	19999	19929	-55.89%	-57.94%
GetNthPrime5000	17105	16564	28893	28568	-68.92%	-72.47%
Foo1000000	15903	15796	25117	24826	-57.94%	-57.17%
ForeignFunctionOverhead1000000	8663	8575	19809	19802	-128.66%	-130.93%
PrimitiveOpOverhead1000000	8432	8412	18927	19351	-124.47%	-130.04%
BoxManipulation	11326	11186	19469	19502	-71.90%	-74.34%
BoxManipulation2	0	0	0	0	#DIV/0!	#DIV/0!

PrimitiveOpOverhead2_1000000	8402	8466	19559	19419	-132.79%	-129.38%
LetTest1000000	6069	6035	9363	9360	-54.28%	-55.10%
ApplyTest	3696	3648	8072	8021	-118.40%	-119.87%

The results show that checking the interrupted status of the thread is very expensive. Perhaps a method for reducing the frequency of checking the interrupt status can be formulated. However, the issues with static member/methods would apply to this as well.

Nov. 19, 2003: Optimization of oversaturated dictionary functions.

Type classes give rise to compiler generated dictionary functions. This gives rise to oversaturated application chains. For example 1 + 2 can give rise to 'add' applied to 'dictionary variable' applied to 1 applied to 2. 'add' is actually an arity 1 supercombinator. Evaluating/resolving the application of 'add' to the 'dictionary variable' results in a function that takes two arguments. In an oversaturated situation where the left hand side will always resolve to a function of know arity, which matches the remaining arguments, the representation/resolution can be optimized in a fashion similar to the optimization applied to fully saturated applications.
Currently there is no way for the code generation to dynamically recognize dictionary functions. However specific knowledge of the methods associated with the Equals, Ord, and Num type classes allows them to be optimized.

	First Run Before	After	% change	Average Before	After	% change
TestBonusRating200	12318	8582	29.3	11607	8389	27.7
BadRegressionGemTest200_200	15823	13860	12.4	15362	13723	10.7
GetNthPrime5000	20490	17445	14.9	19795	17281	12.7
Foo1000000	19188	15373	19.9	19795	15889	16.2
ForeignFunctionOverhead10000000	10455	10525	-0.7	10435	10479	-0.4
PrimitiveOpOverhead10000000	10255	10225	0.3	10261	10311	-0.5
BoxManipulation	13480	10916	19.0	13319	10849	18.5
PrimitiveOpOverhead2_10000000	10195	10125	0.6	10281	10255	0.2
LetTest10000000	7261	5368	26.0	7280	5381	26.1
ApplyTest4000000	4506	4487	0.3	4517	4463	1.2
M2.mainM2	15960	15132	5.2	4853	4339	10.6

November 21, 2003: Overhead of halting lecc execution.

The overhead of checking static flag in the Executor class seems to be minimal based on these results.

	First Run Before	After	% change	Average Before	After	% change
TestBonusRating200	9294	8322	10.5	8495	8302	0.2
BadRegressionGemTest200_200	13790	13750	1.9	13522	13488	1.9
GetNthPrime5000	17095	16985	1.9	16757	16714	1.6
Foo1000000	15433	15031	0.8	15309	15082	-0.3
ForeignFunctionOverhead10000000	10355	10525	-0.05	10361	10482	0.4
PrimitiveOpOverhead10000000	10315	10155	0.3	10278	10281	-1.2

BoxManipulation	10596	10755	-0.3	10669	10692	0.6
PrimitiveOpOverhead2_10000000	10095	10071	-1.4	10244	10171	-1.0
LetTest10000000	5478	5598	0.3	5464	5518	1.4
ApplyTest4000000	4477	4376	0.7	4446	4376	1.6
M2.mainM2	16794	16652	0.8	4376	4299	1.7

December 16, 2003: Overhead of threading an execution context through the runtime code.

Rather than using static flags/members for halting, statistics generation, etc. an execution context can be passed through the runtime code as an extra parameter. Going forward this can be very useful for behaviour that should be specific to an execution session.

The runtime classes were modified to include this extra parameter and the code generation was similarly modified. The halting code and statistics generation were then modified to use the execution context, rather than static members of the Executor class.

The additional overhead of passing this execution context is higher than expected. Unfortunately the need for this context object seems unavoidable which indicates that we will have to accept the performance hit.

	Average Before	After	% change
TestBonusRating200	8238	8803	-6.8
BadRegressionGemTest200_200	13550	14197	-4.4
GetNthPrime5000	17121	18219	-6.4
Foo1000000	15967	16931	-6.0
ForeignFunctionOverhead10000000	12655	14050	-9.6
PrimitiveOpOverhead10000000	12531	13726	-8.9
BoxManipulation	11443	11990	-4.8
PrimitiveOpOverhead2_10000000	12598	13686	-8.6
LetTest10000000	5781	6069	-5.0
ApplyTest4000000	5398	5865	-8.7
M2.mainM2	213	220	-3.3

May 14, 2004: Differentiation between recursive and non-recursive let variables.

The compiler was updated to differentiate between recursive and non-recursive let variables. This allowed the code generated to avoid creating indirection nodes to represent non-recursive let variables. Unfortunately there was no significant performance gain with this change.

There was, however, a decrease in total class file size: with source generation turned on the total class file size went from 20,038,127 bytes to 19,936,477 bytes for a savings of 101,650 bytes. Directly generated byte code went from 21,124,974 bytes to 21,036,403 bytes for a savings of 88,571 bytes. It is interesting to note that the directly generated byte code is 1,099,926 bytes.

Without differentiation		With differentiation			
First Run	Av.	First Run	Avg.	% change 1st run	%change 2nd run

testBonusRating200	4563	3880	4500	3885	1.38%	-0.13%
badRegressionGemTest200_200	6220	5818	6250	5932	-0.48%	-1.96%
getNthPrime5000	7554	7448	8266	7739	-9.43%	-3.91%
foo1000000	7095	7022	7203	7203	-1.52%	-2.58%
foreignFunctionOverhead10000000	6563	6578	6578	6635	-0.23%	-0.87%
primitiveOpOverhead10000000	7454	7568	7422	7495	0.43%	0.96%
boxManipulation	5329	5161	5499	5103	-3.19%	1.12%
boxManipulation2	1938	1890	2016	1947	-4.02%	-3.02%
primitiveOpOverhead2_10000000	7297	7412	7406	7469	-1.49%	-0.77%
letTest10000000	2672	2735	2752	2807	-2.99%	-2.63%
applyTest4000000	2813	2760	2855	2894	-1.49%	-4.86%
M2.mainM2	7047	276	7765	270	-10.19%	2.17%
workbookBenchmark4 (4000)	15235	9052	15313	8910	-0.51%	1.57%

June 29, 2004: Adding strictness annotations to data constructor and function arguments.

Strictness annotations on data constructor and function arguments were added to the language syntax. For functions marked with strict arguments code generation was changed as follows:

- redundant calls to evaluate were removed
- primitive operations in lazy contexts were compiled strictly if both arguments were known to be already evaluated
- better resolution of literal values in primitive operations

Benchmark	Without Strictness	With Strictness	% Change
testBonusRatings800	83621	82986	0.76%
badRegressionGemTest500_500	50841	48516	4.57%
getNthPrime20000	93885	87126	7.20%
foo6000000	60700	52453	13.59%
foreignFunctionOverhead60000000	56309	56029	0.50%
primitiveOpOverhead60000000	55388	55905	-0.93%
box1 (14 levels)	73746	66186	10.25%
box2 (23 levels)	10374	9750	6.02%
primitiveOpOverhead2_60000000	55466	55782	-0.57%
letTest10000000	40561	41594	-2.55%
applyTest60000000	58700	59094	-0.67%
workbookBenchmark4_10000	59965	28132	53.09%
digits_of_e1_800	51498	49312	4.24%
exp3_8	4828	4656	3.56%

July 8, 2004: In-lining of let variables.

Let variables which are only referenced once, or which are defined as a non-reducible entity can be inlined. This has two benefits: 1) It removes the declaration/assignment of a local variable, 2) If the let variable is referenced in a strict context the definition will now be compiled strictly.

Benchmark	Without Inlining	With Inlining	% Change
testBonusRatings800	87375	86533	1.0 %
badRegressionGemTest500_500	51227	50218	2.0%
getNthPrime20000	89762	91565	2.0%
foo6000000	55845	54240	2.9%
foreignFunctionOverhead60000000	57525	58603	-1.87%
primitiveOpOverhead60000000	57609	58731	-1.95 %
box1 (14 levels)	69358	68344	1.46%
box2 (23 levels)	10053	9819	2.33%
primitiveOpOverhead2_60000000	57718	58900	2.05%
letTest10000000	58136	23660	59.30%
applyTest60000000	59414	61094	-2.83%
workbookBenchmark4_10000	27556	27385	0.62%
digits_of_e1_800	54146	50514	6.71%
exp3_8	4785	4629	3.26%

July 21, 2004: Unboxed primitive arguments.

The strict version of the function body is generated to with unboxed arguments where the argument is annotated as strict and is of a primitive type (i.e. int, boolean, double, etc.). The results are encouraging. Some of the more synthetic benchmarks showed a drastic improvement but others, like bonus rating and regression, showed a nice improvement as well.

	No Unboxed Values	Unboxed Primitives	% change
testBonusRatings800	89463	83509	6.66%
badRegressionGemTest500_500	52068	48324	7.19%
getNthPrime20000	90817	87039	4.16%
Foo60000000	55479	52198	5.91%
foreignFunctionOverhead60000000	65995	15425	76.63%
primitiveOpOverhead60000000	67728	15425	77.23%
box1 (14 levels)	76385	65885	13.75%
box2 (12 levels)	10390	9515	8.42%
primitiveOpOverhead2_60000000	62104	15389	75.22%
letTest10000000	44871	21810	51.39%
applyTest60000000	77650	19935	74.33%
workbookBenchmark4	24560	24375	0.75%
Digits_Of_e1_800	49667	48215	2.92%

exp3_8	4281	4265	0.37%

July 28, 2004: Unboxed foreign types and data constructor members.

Unboxing was extended to strict arguments of foreign types. Also data constructor generation was changed so that strict members of foreign or primitive types are held as unboxed values.
This change didn't really create a significant change in any of the existing benchmarks. A greater change will probably be seen as more CAL code is updated with strictness annotations.

	Unboxed primitive arguments only	Unboxed foreign types and DC members	% change
testBonusRatings800	83509	83396	1.35
badRegressionGemTest500_500	48324	48672	-0.72
getNthPrime20000	87039	86904	0.16
Foo60000000	52198	52020	0.34
foreignFunctionOverhead60000000	15425	15500	-0.49
primitiveOpOverhead60000000	15425	15724	-1.94
box1 (14 levels)	65885	63940	2.95
box2 (12 levels)	9515	9351	1.72
primitiveOpOverhead2_60000000	15389	15500	-0.72
letTest10000000	21810	21976	-0.76
applyTest60000000	19935	20219	-1.42
workbookBenchmark4	24375	24671	-1.21
Digits_Of_e1_800	48215	48558	-0.71
exp3_8	4265	4324	-1.38
mainM2	520	502	3.46
Class file size	18914015	18928024	

August 2, 2004: Close examination of getNthPrime.

The getNthPrime benchmark, and all involved supercombinators, was given close examination and various hand-coded optimizations were done.
Timings were done on getNthPrime 10,000.

	Time	% change from previous	%change from base
Base time	30201		
Integer versions of upFrom, upFromThem, and upFromTo.	18848	37.59	37.59
Remove instances of idInt.	18271	3.06	39.50
Create partial application of nonMultiple.	17583	5.01	41.78
Replace fromInt in isPrime with a cast	17354	1.30	42.54

to double.			
Modify upFrom to remove the call to strict.	14954	13.83	50.49
Partial application to unboxed primitives for addInt, >=, and other primitive operations.	14839	0.77	50.87
Changed case expressions on the List data type to be an if-then-else in java.	14724	0.77	51.25

As in previous implementations the biggest single gain was realized by switching from polymorphic to integer versions of upFrom, upFromTo, and upFromThen. The other changes suggest a number of patterns/areas to look at for optimization.

1. In cases where arguments are known to be evaluated laziness can be circumvented in various ways. Strict application nodes can be used instead of lazy ones if all the strict arguments are known to be evaluated. Some specific operations, such as strict or seq, can potentially be removed based on the evaluation state of the arguments.
2. Partial applications can be handled better for some specific supercombinators. For example, partial applications primitive operations can be represented by special nodes that speed up the reduction process.
3. Cases done on data types with two data constructors can be generated as an if-then-else in java. This eliminates the need for a default case.
4. The extraction of data constructor fields in case alternates should be treated like let variables with regards to inlining and removal.
5. Functions which resolve to an identity (like idInt) should be removed, assuming that strictness is not changed by doing so.
6. Converting between types should be investigated for potential optimizations. For example, in isPrime an integer value is implicitly converted to a double value by an application of fromInt. This involves going from an int to a long to a BigInteger, and finally to a double. There should be someway of recognizing that this can be done more efficiently.
7. Expanding on point six, casting in general can be done more efficiently. Currently casting is done through application of static foreign functions. For the lecc machine these casts could be done more efficiently in the generated java source code via the java cast mechanism or be removed entirely.

August 4, 2004: Optimizations to cases on two DC data types and extended use of strict application nodes.

Case statements on data types with exactly two data constructors were generated as an if-then-else in the java code.
When a fully saturated supercombinator application is encountered in a lazy context a strict application node is now generated if all the strict arguments to the supercombinator are known to be evaluated.

The biggest change was for the let variable benchmark. However, after modifying the definition of letTest to include a type signature for a local function the time went down to 6647 ms. This is a gain of 66.85%, indicating that it's a good idea to include type signatures to narrow the type of a function whenever possible.

	Before optimizations	After optimizations	% change

testBonusRatings800	83396	83740	-0.41
badRegressionGemTest500_500	48672	47337	2.74
getNthPrime20000	86904	85902	1.15
Foo60000000	52020	50802	2.34
foreignFunctionOverhead60000000	15500	15112	2.50
primitiveOpOverhead60000000	15724	15113	3.89
box1 (14 levels)	63940	64853	-1.42
box2 (12 levels)	9351	9324	0.29
primitiveOpOverhead2_60000000	15500	15312	1.21
letTest10000000	21976	20052	8.76
applyTest60000000	20219	19792	2.11
workbookBenchmark4	24671	24594	0.31
Digits_Of_e1_800	48558	48390	0.35
exp3_8	4324	4183	3.26
mainM2	502	515	-2.59
Class file size	18928024	19453224	

August 27, 2004: Changes to core Prelude functions to take advantage of un-boxing and strictness optimizations.

These benchmarks show changes caused primarily by re-writing core functions in the Prelude module to take advantage of un-boxing and strictness optimizations. There are also a few other optimizations included. These primarily affect the handling of literal values and doing speculative evaluation of primitive and foreign functions if all arguments are known to be evaluated.

It is quite obvious that the un-boxing and strictness optimizations can have a huge benefit. However, this benefit only occurs if the CAL code is written in such a way as to allow optimizations. Updated versions of core functions have been up to five times as fast as the original versions. To see the benefits of un-boxing and strictness more generally guidelines for CAL coders, on how to write high-performance code, need to be generated and distributed.

	Before optimizations	After optimizations	% change
testBonusRatings800	83740	46500	44.47
badRegressionGemTest500_500	47337	17219	63.62
getNthPrime20000	85902	20531	76.10
Foo60000000	50802	7219	85.79
foreignFunctionOverhead60000000	15112	12422	17.80
primitiveOpOverhead60000000	15113	12703	15.95
box1 (14 levels)	64853	25875	60.10
box2 (12 levels)	9324	5500	41.01
primitiveOpOverhead2_60000000	15312	12734	16.83
letTest10000000	20052	18907	5.71
applyTest60000000	19792	16953	14.34
workbookBenchmark4	24594	24891	-1.21
Digits_Of_e1_800	48390	17234	64.39
exp3_8	4183	3797	9.23

September 1, 2004: Conversion of tail recursive functions into loops.

This optimization converts tail recursive functions into loops by introducing a 'while (true)' loop and a continuation at the point of the tail recursive call. For some benchmarks this made a significant difference, even to the point of making the benchmark useless going forward. (E.g. primitiveOpOverhead).
The other really significant gain was on the list traversal benchmarks. This is encouraging because iterating over a data structure is a common pattern in our existing CAL code.
Total class file size changed from 20,192,880 to 20,341,799 with this optimization, for an increase of 0.74%.

	Client VM			Server VM			% Diff. client vs. server	
	No-loops	Loops	% change	No-Loops	Loops	% change	No-Loops	Loops
testBonusRatings800	50853	48442	4.74%	38634	36448	5.66%	24.03%	24.76%
badRegressionGemTest500_500	18135	15532	14.35%	13561	11794	13.03%	25.22%	24.07%
getNthPrime20000	22463	20204	10.06%	18125	14776	18.48%	19.31%	26.87%
foo60000000	8224	6455	21.51%	6439	4708	26.88%	21.70%	27.06%
foreignFunctionOverhead60000000	15442	84	99.46%	11582	36	99.69%	25.00%	57.14%
primitiveOpOverhead60000000	15458	78	99.50%	11763	47	99.60%	23.90%	39.74%
box1 (14 levels)	29755	26967	9.37%	24320	22699	6.67%	18.27%	15.83%
box2 (12 levels)	5776	5496	4.85%	4449	4039	9.22%	22.97%	26.51%
primitiveOpOverhead2_60000000	15385	109	99.29%	11693	109	99.07%	24.00%	0.00%
letTest10000000	21619	18433	14.74%	15380	18705	-21.62%	28.86%	-1.48%
applyTest60000000	19906	20011	-0.53%	15755	15860	-0.67%	20.85%	20.74%
listTraversalLast	61463	14947	75.68%	53616	12667	76.37%	12.77%	15.25%
listTraversalSubscript	64775	14402	77.77%	51940	12272	76.37%	19.81%	14.79%
workbookBenchmark4	23906	24261	-1.48%	21440	21147	1.37%	10.32%	12.84%
digits_of_e1_800	18036	17739	1.65%	15198	14335	5.68%	15.74%	19.19%
exp3_8	4099	4132	-0.81%	3050	2945	3.44%	25.59%	28.73%

September 10, 2004: Comparison between generated source and generated byte code with tail recursion optimization.

This is a performance comparison between the generated source code and the generated byte code after updating byte code generation to handle the tail recursion optimization.
Interestingly the generated byte code is quite consistently slower (in the 1 – 3% range) than the generated source code. On some benchmarks the generated byte code is a lot slower (i.e. foreign function and primitive operations) but these are benchmarks that run so quickly that other external factors can have an overriding effect.
The exceptions to the byte code being slower were the list traversal benchmarks. On the server JVM these were both faster.

	Client VM			Server VM			% Diff. client vs. server	
	Source	Byte	% change	Source	Byte	% change	Source	Byte
testBonusRatings800	48176	49053	-1.82%	38676	39149	-1.22%	19.72%	20.19%
badRegressionGemTest500_500	15158	15715	-3.67%	12005	12264	-2.16%	20.80%	21.96%

getNthPrime20000	19486	20049	-2.89%	15641	15604	0.24%	19.73%	22.17%
foo60000000	6188	6245	-0.92%	4880	5033	-3.14%	21.14%	19.41%
foreignFunctionOverhead60000000	109	94	13.76%	36	47	-30.56%	66.97%	50.00%
primitiveOpOverhead60000000	79	93	-17.72%	36	42	-16.67%	54.43%	54.84%
box1 (14 levels)	26268	27003	-2.80%	21187	21800	-2.89%	19.34%	19.27%
box2 (12 levels)	5438	5520	-1.51%	4093	4116	-0.56%	24.73%	25.43%
primitiveOpOverhead2_60000000	110	109	0.91%	104	109	-4.81%	5.45%	0.00%
letTest10000000	18433	18658	-1.22%	21315	22030	-3.35%	-15.64%	-18.07%
applyTest60000000	22774	23663	-3.90%	17590	18015	-2.42%	22.76%	23.87%
listTraversalLast	9820	10017	-2.01%	7966	7482	6.08%	18.88%	25.31%
listTraversalSubscript	9111	9532	-4.62%	8163	7774	4.77%	10.41%	18.44%
workbookBenchmark4	24347	24660	-1.29%	20966	21039	-0.35%	13.89%	14.68%
digits_of_e1_800	17633	18259	-3.55%	14518	14964	-3.07%	17.67%	18.05%
exp3_8	4132	4191	-1.43%	3038	3125	-2.86%	26.48%	25.44%

September 28, 2004: Addition of several optimizations.

A variety of different optimizations were added. These included:
- Optimizing class methods by in-lining the application of the dictionary argument to the dictionary index.
- Treating enumeration data types as primitive integers for purposes of strictness and un-boxing.
- Optimizations to switching on integers and Booleans when dealing with unboxed values.

	Client VM			Server VM			% Diff. client vs. server	
	Before	After	% change	Before	After	% change	Before	After
testBonusRatings800	48176	47294	1.83%	38676	37703	2.52%	19.72%	20.28%
badRegressionGemTest500_500	15158	13237	12.67%	12005	9195	23.41%	20.80%	30.54%
getNthPrime20000	19486	19864	-1.94%	15641	16008	-2.35%	19.73%	19.41%
foo60000000	6188	6000	3.04%	4880	5122	-4.96%	21.14%	14.63%
box1 (14 levels)	26268	25471	3.03%	21187	21037	0.71%	19.34%	17.41%
box2 (12 levels)	5438	3728	31.45%	4093	3086	24.60%	24.73%	17.22%
letTest10000000	21861	27601	-26.26%	21315	27772	-30.29%	2.50%	-0.62%
applyTest60000000	22774	21272	6.60%	17590	17043	3.11%	22.76%	19.88%
workbookBenchmark4	24347	21790	10.50%	20966	19611	6.46%	13.89%	10.00%
digits_of_e1_800	17633	18077	-2.52%	14518	15742	-8.43%	17.67%	12.92%
exp3_8	4132	3901	5.59%	3038	3133	-3.13%	26.48%	19.69%

There were mixed results. The regression and box2 benchmarks showed good improvements. However the let benchmark showed degradation in performance of about 30%. Investigation showed that the optimization to in-line the application of the dictionary function when dealing with class methods removed a runtime specific optimization in the lecc machine. The lecc machine was recognizing applications of specific class methods and creating an optimized graph representation for the oversaturated application chain.

The lecc optimization to was modified to recognize the new pattern of applying a dictionary function argument to an integer index value in an attempt to restore the lost performance. Interestingly the results were better than expected. The time for letTest went from 27 seconds to 9.7 seconds. The combination of the dictionary optimization from the compiler and the lecc optimization reduced the number of reductions and node allocations by 66% for the letTest benchmark. As well, the time of digits_of_e1 came back into line with previous results.

November 18, 2004: Doing tail calls on the java stack in LECC.

Previously non-recursive tail calls were always implemented by creating a strict application node, representing the tail call, and returning it to the evaluation loop. However, it is safe to simply make the tail call on the java stack if the function being called is not strongly connected to the calling function and the function being called is not itself tail recursive.

Benchmark	Before (ms)	After (ms)	% change
testBonusRatings800	24078	19422	19.33
badRegressionGemTest500_500	6844	6625	3.20
getNthPrime20000	10359	9687	6.49
foo60000000	3375	3281	2.79
box1 (14 levels)	15093	14932	1.07
box2 (12 levels)	1984	1907	3.88
letTest10000000	4669	4578	1.95
applyTest60000000	13703	13719	-0.12
M2.mainM2	2015	1875	6.95
workbookBenchmark4	12984	12718	2.07
digits_of_e1_800	11719	11797	-0.67
Exp3_8	2297	2312	-0.65

The largest gains were in the bonus rating gems, get Nth prime gems, and M2.mainM2. The rest of the benchmarks showed no significant change.

May 17, 2005: Effects of volatile quit flag in lecc.

Comparing a build from the last time this document was updated to a current build, as of May 12, the majority of the benchmarks have suffered a significant degradation in performance. Looking at the weekly builds during the intervening period it was determined that the biggest single change that caused a performance hit was changing the quit flag in RTExecutionContext to be volatile. This change was necessary for correctness, since without it the evaluation thread might not see a change to this flag made by a client thread.

Benchmark	Build from Nov. 25, 2004	May 12, 2005	% Change
testBonusRatings800	23953	24125	-0.7
badRegressionGemTest500_500	6719	7453	-10.9
getNthPrime20000	9125	10578	-15.9
foo60000000	3469	4156	-19.8
box1 (14 levels)	13015	14485	-11.29
box2 (12 levels)	1937	2094	-8.1

letTest10000000	4578	5203	-13.65
applyTest60000000	13719	16172	-17.88
M2.mainM2	1829	1328	27.4
workbookBenchmark4	12672	12578	0.74
digits_of_e1_800	11547	12515	-8.38
Exp3_8	2313	2906	-25.63

A few different things were tried to ameliorate the performance hit caused by changing the quit flag to volatile. Making the flag non-volatile and synchronizing the access functions made things worse for all but one benchmark.

Benchmark	May 12, 2005	Synchronize quit flag access fctns.	% Change
testBonusRatings800	24125	24506	-1.6
badRegressionGemTest500_500	7453	9188	-23.28
getNthPrime20000	10578	11500	-8.72
foo60000000	4156	4718	-13.52
box1 (14 levels)	14485	15281	-5.50
box2 (12 levels)	2094	2188	-4.49
letTest10000000	5203	5985	-15.03
applyTest60000000	16172	13781	14.78
M2.mainM2	1328	1360	-2.41
workbookBenchmark4	12578	12750	-1.37
digits_of_e1_800	12515	13344	-6.62
Exp3_8	2906	3109	-6.99

The access functions in RTExecutionContext were modified so that the volatile flag is only accessed every nth time that the function is called.

Benchmark	May 12, 2005	Check flag every 2nd time	% Change from previous	Check flag every 5th time	% Change from previous	Check flag every 10th time	% Change from previous
testBonusRatings800	24125	22187	8.03	21219	4.36	20921	1.4
badRegressionGemTest 500_500	7453	6578	11.74	6560	0.27	6500	10.74
getNthPrime20000	10578	9890	6.50	9188	7.10	9250	-0.67
foo60000000	4156	3578	13.91	3640	-1.73	3407	6.40
box1 (14 levels)	14485	14313	1.19	13360	6.66	13235	0.94
box2 (12 levels)	2094	2000	4.49	1985	0.75	1890	4.79
letTest10000000	5203	4890	6.02	4703	3.82	4641	1.32
applyTest60000000	16172	13891	14.10	13671	1.58	13594	0.56
M2.mainM2	1328	1234	7.08	1156	6.32	1172	-1.38
workbookBenchmark4	12578	12438	1.11	12422	0.13	12375	0.38
digits_of_e1_800	12515	11344	9.36	11712	-3.2	10703	8.62
Exp3_8	2906	2531	12.90	2453	3.08	2406	1.92

May 17, 2005: Strictness bug fix.

A space usage bug was encountered. Given the following two functions, the performance and space use characteristics should be essentially the same. However, when running the two functions with input of (upFromTo 1.0 10000000) average4 completed successfully but average6 ran out of memory.

```
average4 :: [Double] -> Double;
average4 !xs = 'average 4 definition'.

average6 :: (Num a, Typeable a) => [a] -> Double;
average6 !xs =
    average4 (unsafeCoerce xs);
```

Investigation determined that there were two bugs associated with strictness optimizations.
The first bug was with the optimizations where tail calls were made directly on the java call stack, rather than building/returning a closure. If the stack frame of the calling function holds a reference to the root of an argument to the tail call we can get incorrect space usage behaviour if the argument is a data type that is expanded in the evaluation of the tail call.
The second bug is similar to the first but applies to the eager evaluation of strict arguments to a tail call in order to build a strict application node.

Timings with the optimizations turned off showed an expected degradation in performance. Rather than simply eliminate both optimizations code generation was changed to use strict application nodes for tail calls under a more restrictive set of circumstances. Rather than eagerly evaluate strict arguments in order to build a strict application node we know examine the strict argument values and build a strict application node if all strict arguments are known to be evaluated, or if they are fully saturated applications of known safe functions for which all arguments are known to be evaluated.
The following table shows the results of turning off these two optimizations and then partially restoring the use of strict application nodes.

Benchmark	Before changes.	Optimizations off.	% Change from previous	Use strict node when possible	% Change from previous	% Change from before changes.
testBonusRatings800	20921	27171	-29.87	24093	11.33	-15.16
badRegressionGemTest 500_500	6500	7281	-12.02	6485	10.93	0.23
getNthPrime20000	9250	10906	-17.09	9922	9.02	-7.26
foo60000000	3407	3469	-1.82	3562	-2.68	-4.54
box1 (14 levels)	13235	14344	-8.38	13500	5.88	-2.00
box2 (12 levels)	1890	2360	-24.87	2219	5.97	-17.98
letTest10000000	4641	5281	-13.79	5329	-0.91	-14.82
applyTest60000000	13594	29062	-113.79	16484	43.28	-21.26
M2.mainM2	1172	1391	-18.69	1187	14.67	-1.28
workbookBenchmark4	12375	13062	-5.55	12781	2.15	-3.28
digits_of_e1_800	10703	11515	-7.59	10719	6.91	-0.15
Exp3_8	2406	2437	-1.29	2422	0.62	-0.67
SimpleRolap2	7969	9438	-18.43	8031	14.91	-0.78

Restoring the use of strict application nodes under stricter requirements does ameliorate some of the losses caused by turning off the optimizations that were causing a problem. However there is still an overall decrease in performance which is particularly severe for the box2, letTest, and applyTest benchmarks.

May 19, 2005: Checking for quit flag inside tail recursion loops in lecc.

One optimization in the lecc machine is to turn tail recursive functions into loops. However, checking the status of the quit flag was only done when a 'evaluate' was called on a graph node. This could lead to a lack of responsiveness if the body of a tail recursion loop didn't make any calls to evaluate.
The code generation was modified to include a check on the quit flag in the body of the tail recursion loop.
(Note: These benchmarks are done on a different machine than the May 17th entry in this document. This accounts for the significant change in run times.)

Benchmark	Before new quit check.	After.	% Change from previous
testBonusRatings800	65475	65190	0.43
badRegressionGemTest 500_500	16013	17115	-6.88
getNthPrime20000	25937	26080	-0.55
foo60000000	7421	7551	-1.75
box1 (14 levels)	31084	31308	-0.72
box2 (12 levels)	5058	5008	0.99
letTest10000000	12009	12219	-1.75
applyTest60000000	32607	32710	-0.32
M2.mainM2	9354	9544	-2.03
workbookBenchmark4	31090	30376	2.30
digits_of_e1_800	20689	20842	-0.74
Exp3_8	5568	5639	-1.28
SimpleRolap2	20860	21003	-0.69

June 7, 2005: Restore use of strict application nodes for tail calls in lecc machine.

Previously the eager evaluation of strict arguments to tail calls, which allowed the use of strict application nodes, was turned off due to a space usage issue. (See change from May 17, 2005.)
After investigating the situations in which the space usage problem could be caused by eager evaluation of strict arguments to a tail call and considering the behaviour of other languages/runtimes (clean, ghc, hugs) it was decided that the eager evaluation of strict arguments was worth the space usage issue. Accordingly this is being turned back on. Note: this does not restore performing tail calls directly on the java call-stack; it only restores the eager evaluation of strict arguments in tail class in order to use strict application nodes.

It was determined that the memory issue with eager argument evaluation comes down to situations where:

- a let variable is the root of a dynamically generated self-referential data structure (such as a list)
- the let variable is not in-lined
- the let variable is used in the expression for an eagerly evaluated argument in a tail call

- evaluating the argument causes the data structure created by the let variable to be expanded

For example:

```
Foo =
  let x = upFromTo 1.0 1000000.0 in (head x) + (last x);
```

The two arguments to the tail call (in this case addDouble) will be eagerly evaluated. Because the let variable is not in-lined there is a reference to the head of the list which is held in the stack from for Foo. This results in the entire list residing in memory, instead of being discarded as it is traversed by last. If the list is sufficiently large performance problems or an out-of-memory error can result. Hugs and ghc run this function in constant space while in CAL and Clean the memory usage is directly proportional to the length of the list.

Interestingly enough a similar example:

```
Foo =
  let x = upFromTo 1.0 1000000.0 in (last x) + (head x);
```

This example uses memory proportional to list length in Hugs, ghc, and CAL, while in Clean in runs in constant space. For Clean, Hugs, and ghc the order of evaluation of the arguments determines space behaviour. i.e. if 'head x' is evaluated before 'last x' then space use is constant. This does make sense, because if 'last x' is evaluated first the root of the list is still being held in the expression 'head x' causing the entire list to exist in memory. We can also determine that the order of argument evaluation is reversed in Clean from what it is in Hugs/ghc.

To fully match the space behaviour of the other languages CAL would have to do turn off eager argument evaluation. This would mean that primitive ops and foreign functions would never be directly evaluated but would always create a call to the corresponding functional form. Also, all case statements and if-then-else constructs would have to be lifted into separate functions. These changes would present a serious degradation in performance. Given that the set of circumstances where the space behaviour problem occurs is very rarely encountered in production code (it occurs more often in test code where lists are dynamically created in order to work with large data structures) it was decided to leave CAL space behaviour as it is currently.

(Note: These benchmarks numbers were generated on a different machine than the May 19th entry in this document. This accounts for the significant change in run times.)

	Base	Add back eager eval.	% change
testBonusRatings800	28328	24217	14.51%
badRegressionGemTest500_500	7765	6610	14.87%
getNthPrime20000	11265	9617	14.63%
foo60000000	3547	3468	2.23%
Foreign Function Overhead	563	570	-1.24%
Primitive op overhead	578	570	1.38%
box1 (14 levels)	14796	13345	9.81%
box2 (12 levels)	2313	1819	21.36%
Primitive op overhead 2	609	601	1.31%
letTest10000000	5281	4793	9.24%
applyTest60000000	16766	14085	15.99%

Average	3484	2820	19.06%
Average on expanded list	4578	4083	10.81%
Partial applications	28514	26389	7.45%
M2.mainM2	1532	1195	22.00%
workbookBenchmark4	11203	10369	7.44%
digits_of_e1_800	11391	10789	5.28%
Exp3_8	2453	2391	2.53%
SimpleRolap2	9124	8763	3.96%

June 7, 2005: Optimistic reduction in lecc machine.

When evaluating a graph a loop is entered which keeps calling 'reduce' on the graph head as long as the graph is reducible. Testing to determine the whether the graph can be reduced involves walking the nodes in the graph to determine the number of arguments applied to, and the arity of, the supercombinator on the left hand side of the graph.

Rather than walking the graph to determine its reducibility each time through the loop. The node classes were modified so that if a graph is not reducible it will simply return its own head. The loop was then modified to run as long as the result of calling 'reduce' on the current graph head doesn't return the head itself.

	Base	Optimistic reduction	% change
testBonusRatings800	24217	20593	14.96%
badRegressionGemTest500_500	6610	5656	14.43%
getNthPrime20000	9617	8046	16.34%
foo60000000	3468	3375	2.68%
Foreign Function Overhead	570	625	-9.65%
Primitive op overhead	570	625	-9.65%
box1 (14 levels)	13345	11484	13.95%
box2 (12 levels)	1819	1719	5.50%
Primitive op overhead 2	601	657	-9.32%
letTest10000000	4793	4735	1.21%
applyTest60000000	14085	9687	31.22%
Average	2820	2266	19.65%
Average on expanded list	4083	3594	11.98%
Partial applications	26389	19718	25.28%
M2.mainM2	1195	1016	14.98%
workbookBenchmark4	10369	9796	5.53%
digits_of_e1_800	10789	10703	0.80%
Exp3_8	2391	2406	-0.63%
SimpleRolap2	8763	6859	21.73%

June 9, 2005: Direct Creation of data constructors in lazy contexts.

When a fully saturated application of a data constructor is encountered in a lazy context the lecc machine was simply creating an application of the functional form of the data constructor. However, if the data constructor contains no strict fields, or the values for all strict fields have already been evaluated, it is safe to directly create the instance of the data constructor.
The impact of the change on most of the benchmarks was negligible. However, the SimpleRolap2 benchmark shows an improvement of almost 6%.

	Base	Direct creation of DCs	% change
testBonusRatings800	20593	20186	1.98%
badRegressionGemTest500_500	5656	5484	3.04%
getNthPrime20000	8046	7859	2.32%
foo60000000	3375	3375	0.00%
box1 (14 levels)	11484	11530	-0.40%
box2 (12 levels)	1719	1726	-0.41%
letTest10000000	4735	4750	-0.32%
applyTest60000000	9687	9703	-0.17%
Average	2266	2250	0.71%
Average on expanded list	3594	3594	0.00%
Partial applications	19718	19867	-0.76%
M2.mainM2	1016	1016	0.00%
workbookBenchmark4	9796	9859	-0.64%
digits_of_e1_800	10703	10718	-0.14%
Exp3_8	2406	2422	-0.67%
SimpleRolap2	6859	6468	5.70%

June 20, 2005: Enhancements to lecc reduction algorithm for better handling of oversaturated application chains.

A series of changes was made to enhance the reduction of oversaturated application chains.
The first table shows the benchmark timings as the series of changes was applied, with a percentage change over the previous times.
The second table shows the cumulative change over the original times for the enhancements that were retained.

Change 1: Smarter compilation of the left-hand-side of an oversaturated application chain.
When generating code for application chains where the left-hand-side is a known supercombinator the lecc machine broke to chain into two categories. 1) Exactly the correct number of arguments for the supercombinator were present, 2) there were too few or too many arguments. In case 1 a special purpose application node representing a fully saturated application is created (e.g. RTApp2L). In case 2 the generated code simply used the RTValue.apply method build an application chain of general purpose RTApplication nodes. However, if the application chain is over saturated at some point down the left-hand-side of the chain

there exists a fully saturated application. It is better if the fully saturated application is represented by one of the special purpose nodes and the result then applied using general purpose application nodes.
This change produced some good improvements, especially for the bonus rating and regression gem benchmarks.

Change 2: Introduction of an application node specific to oversaturation.
When reducing a general purpose application node the first thing done is to check if the application graph is oversaturated (i.e. there are more arguments than the supercombinator requires). If this is the case the graph is walked to the left until the root of the fully saturated application is found and that sub-graph is reduced. This continues until the original graph no longer represents an over-saturation.
If the root application node knows that it is the root of an oversaturated application it wouldn't need to perform the original check, which involves walking the graph determining the number of arguments and the supercombinator. Instead the node could simply evaluate its left-hand-side and then continue knowing that it was no longer the root of an oversaturated application.
A new graph node was derived from RTApplication. RTApplicationO is a general purpose application node which is created when we can determine that the left hand side of the application is already fully saturated. It takes advantage of this knowledge to optimize the reduce method. In order to actually create instances of this new node at runtime the apply method was overridden in RTApp, RTCaf, and RTApplicationO. The overridden version of apply produces and instance of RTApplicationO (rather than RTApplication) because in these cases we know that the left-hand-side of the new application is already fully (or over) saturated.
This change had a significant impact on only one of the benchmarks: SimpleRolap2. However, given that SimpleRolap2 is considered representative of a whole class of 'real world' code this is still a good result.

Change 3: Smarter code generation for oversaturated application chains.
After change 1 an oversaturated application chain was being compiled to a smarter set of graph nodes. However a further optimization could be included by differentiating between strict and lazy contexts when generating code for the left-hand-side of the oversaturated chain. In the case of a strict context (where we know the application chain is going to immediately be reduced) we can generate code that directly evaluates the left-hand-side and then applies the result the right-hand-side.
This change only showed significant improvement for two benchmarks: bonus rating and simple rolap 2. Given that these two benchmarks are derived from examples of 'real world' CAL code this is still a good result.

Change 4: Avoid multiple graph traversals when reducing.
When reducing a graph the graph is traversed multiple times for different purposes. It is traversed once to get the supercombinator on the left-hand-end of the graph. A second traversal occurs to count the number of arguments applied to the supercombinator. If the number of arguments is correct for the supercombinator the body of the supercombinator is then invoked and passed the graph root as an argument. The first thing that the supercombinator body does is traverse the graph a third time to extract the argument values.
The reduce method in RTApplication/RTApplicationO was modified so that the graph was only traversed once. As the graph was traversed to the left the argument values were placed in an RTValue array. At the end of the traversal the supercombinator was known (being the leftmost node), the number of arguments was known, and all the argument values had been extracted and placed in the array. The generated code for supercombinator bodies was changed to take an RTValue array containing the argument values.
The RTValue array which was used to contain argument values was allocated as a field in the RTExecutionContext instance passed through the runtime reduction. The array could be re-used as the

supercombinator bodies extracted all argument values from the array into a local variable before executing any function logic.
Unfortunately this change had no discernable positive impact, and in fact slowed down some of the benchmarks noticeable. As a result this change was not retained.

As can be seen in the second table the overall results of changes 1-3 were quite positive. Of special note is the improvement for the bonus rating, bad regression gem, and simple rolap 2 benchmarks. Since these are all benchmarks derived from CAL code that was written to solve a 'real world' problem and had been included in the benchmarks because they proved to have problematic performance it is nice to see them respond so positively to an enhancement.

	Base	Change 1	% Change	Change 2	% Change	Change 3	% Change	Change 4	% Change
testBonusRatings800	21156	19828	6.28%	19624	1.03%	18938	3.50%	19187	-1.31%
badRegressionGemTest 500_500	6578	5656	14.02%	5546	1.94%	5578	-0.58%	5610	-0.57%
getNthPrime20000	8234	7922	3.79%	8078	-1.97%	8156	-0.97%	8203	-0.58%
foo60000000	3406	3375	0.91%	3360	0.44%	3360	0.00%	3360	0.00%
box1 (14 levels)	12094	11895	1.65%	11687	1.75%	11656	0.27%	11812	-1.34%
box2 (12 levels)	1766	1719	2.66%	1750	-1.80%	1750	0.00%	1750	0.00%
letTest10000000	4953	4703	5.05%	4781	-1.66%	4713	1.42%	4703	0.21%
applyTest60000000	9797	9656	1.44%	9671	-0.16%	9661	0.10%	9656	0.05%
Average	2313	2235	3.37%	2203	1.43%	2224	-0.95%	2282	-2.61%
Average on expanded list	3687	3563	3.36%	3531	0.90%	3552	-0.59%	3594	-1.18%
Partial applications	20297	19703	2.93%	19859	-0.79%	19535	1.63%	20640	-5.66%
M2.mainM2	1094	1078	1.46%	1063	1.39%	1032	2.92%	1062	-2.91%
workbookBenchmark4	9984	9891	0.93%	9813	0.79%	9953	-1.43%	9859	0.94%
scorecard benchmark	531	516	2.82%	516	0.00%	531	-2.91%	547	-3.01%
digits_of_e1_800	11188	10750	3.91%	10734	0.15%	10828	-0.88%	10875	-0.43%
Exp3_8	2407	2390	0.71%	2390	0.00%	2406	-0.67%	2391	0.62%
SimpleRolap2	6672	6578	1.41%	6187	5.94%	5875	5.04%	6031	-2.66%

	Base	Changes 1-3	% change
testBonusRatings800	21156	18938	10.48%
badRegressionGemTest500_500	6578	5578	15.20%
getNthPrime20000	8234	8156	0.95%
foo60000000	3406	3360	1.35%
box1 (14 levels)	12094	11656	3.62%
box2 (12 levels)	1766	1750	0.91%
letTest10000000	4953	4713	4.85%
applyTest60000000	9797	9661	1.39%
Average	2313	2224	3.85%
Average on expanded list	3687	3552	3.66%

Partial applications	20297	19535	3.75%
M2.mainM2	1094	1032	5.67%
workbookBenchmark4	9984	9953	0.31%
scorecard benchmark	531	531	0.00%
digits_of_e1_800	11188	10828	3.22%
Exp3_8	2407	2406	0.04%
SimpleRolap2	6672	5875	11.95%

July 6, 2005: Caching of boxed values in lecc.

Following a pattern introduced in the standard java libraries in JDK 1.5 the lecc was modified to cache boxed values. Boxed values of type byte (-128 to 127), char (0 to 127), int (-128 to 127), long (-128 to 127), and short (-128 to 127) are now cached.

The change had two steps: 1) modify the generated code to create instances of boxed values through factory methods, rather than simply invoking the constructor. 2) Change the boxing classes to implement a cache and initialize/access it as necessary.

Benchmark numbers are given for each of the two steps and were generated using both the client and server virtual machines.

The client virtual machine shows little overall change, except for the box manipulation benchmarks. Both of these benchmarks involve creating many instances of boxed integers in the range 1-20, as a result they show significant improvement. The decrease in performance caused by using a static method rather than directly invoking the constructor is surprising, as in-lining would be expected to occur and eliminate any penalty.

The numbers from the server virtual machine are interesting in that they show a more consistent overall improvement across the benchmark suite. The obvious exception is the scorecard benchmark which shows a 22% decrease in overall performance. However, repeated benchmarking showed a wide variation in timings with the server virtual machine. The variations were as much as 30-40% so the displayed results aren't of great concern.

Client JVM:

	baseline	use make	% change	cached RTKernel instances	% chagne	total % chagne
testBonusRatings800	19828	25703	-29.63%	20109	21.76%	-1.42%
badRegressionGemTest500_500	5656	6985	-23.50%	5703	18.35%	-0.83%
getNthPrime20000	8093	10125	-25.11%	8407	16.97%	-3.88%
foo60000000	3422	3609	-5.46%	3406	5.62%	0.47%
box1 (14 levels)	12125	13609	-12.24%	10609	22.04%	12.50%
box2 (12 levels)	1766	1890	-7.02%	1547	18.15%	12.40%
letTest10000000	4766	4781	-0.31%	4781	0.00%	-0.31%
applyTest60000000	9750	9766	-0.16%	9781	-0.15%	-0.32%

Average	2234	2953	-32.18%	2250	23.81%	-0.72%
Average on expanded list	3563	4390	-23.21%	3563	18.84%	0.00%
Partial applications	20453	28968	-41.63%	20984	27.56%	-2.60%
M2.mainM2	1140	1391	-22.02%	1141	17.97%	-0.09%
workbookBenchmark4	10313	10344	-0.30%	10281	0.61%	0.31%
scorecard benchmark	547	547	0.00%	546	0.18%	0.18%
digits_of_e1_800	11188	10953	2.10%	11000	-0.43%	1.68%
Exp3_8	2421	2422	-0.04%	2422	0.00%	-0.04%
SimpleRolap2	5906	6750	-14.29%	5985	11.33%	-1.34%
SME	158684	189419	-19.37%	156059	17.61%	1.65%

Server JVM:

	baseline	use make	% change	cached RTKernel instances	% chagne	total % chagne
testBonusRatings800	17359	19459	-12.10%	15696	19.34%	9.58%
badRegressionGemTest500_500	4784	5311	-11.02%	4217	20.60%	11.85%
getNthPrime20000	7230	7544	-4.34%	6638	12.01%	8.19%
foo60000000	2856	2671	6.48%	2671	0.00%	6.48%
box1 (14 levels)	7241	7747	-6.99%	6950	10.29%	4.02%
box2 (12 levels)	1483	1406	5.19%	1234	12.23%	16.79%
letTest10000000	3652	3624	0.77%	3561	1.74%	2.49%
applyTest60000000	7085	6888	2.78%	7028	-2.03%	0.80%
Average	1966	2015	-2.49%	1687	16.28%	14.19%
Average on expanded list	2965	3046	-2.73%	2624	13.85%	11.50%
Partial applications	16152	16602	-2.79%	14416	13.17%	10.75%
M2.mainM2	889	1156	-30.03%	906	21.63%	-1.91%
workbookBenchmark4	9082	9152	-0.77%	9199	-0.51%	-1.29%
scorecard benchmark	562	703	-25.09%	687	2.28%	-22.24%
digits_of_e1_800	10180	9964	2.12%	9480	4.86%	6.88%
Exp3_8	2061	1968	4.51%	1937	1.58%	6.02%
SimpleRolap2	5012	4935	1.54%	4358	11.69%	13.05%
SME	149544	149311	0.16%	130507	12.59%	12.73%

August 11, 2005: Lifting of complex expressions in lazy contexts.

Expressions encountered in a lazy context which involved primitive operations, a conditional, or boolean '&&' or '||' were lifted into their own function. This was done at the code generation level in the lecc machine. An inner class was generated for each lifted expression which derived from RTResultFunction and encapsulated the logic of the expression.

Tests of M2.getNthPrime4 and M2.getNthPrime5 from before and after the optimization were compared. getNthPrime5 differs from getNthPrime4 in that this optimization has been manually applied to the expression '`(Prelude.rem nToTest listHead != (0 :: Int))`'. The results show that the 22% difference in speed between the two functions is removed with the application of the lifted expression optimization.

	getNthPrime4	getNthPrime5	% Difference
Before lifting	2953	2297	22.21%
After lifting	2263	2295	-1.41%

Unfortunately the standard benchmark suite didn't show any significant improvements from this optimization. The biggest change was in the box2 benchmark which showed an 8% improvement, followed by the workbook benchmark at 5%.

	Base	Lifted Expressions	% Difference
testBonusRatings800	19779	19421	1.81%
badRegressionGemTest500_500	5515	5516	-0.02%
getNthPrime20000	7749	7750	-0.01%
foo60000000	3468	3438	0.87%
box1 (14 levels)	10749	10656	0.87%
box2 (12 levels)	1687	1547	8.30%
letTest10000000	4843	4828	0.31%
applyTest60000000	9713	9519	2.00%
Average	2328	2313	0.64%
Average on expanded list	3625	3656	-0.86%
Partial applications	19764	20093	-1.66%
M2.mainM2	1396	1452	-4.01%
workbookBenchmark4	9843	9334	5.17%
scorecard benchmark	546	531	2.75%
digits_of_e1_800	10590	10566	0.23%
Exp3_8	2406	2388	0.75%
SimpleRolap2	7594	7507	1.15%
SME	161397	159130	1.40%

September 19, 2005: Bit operations as built in primitives.

The CAL typeclass Bits defines a set of bitwise operations. Currently the only two implementing types are Prelude.Int and Prelude.Long. The class methods for these two types were implemented as foreign functions which used the appropriate java operators (i.e. &, |, ^, <<, >>, >>>).
Primitive operations were added for the bitwise operations for Int and Long, which replaced the foreign functions. As can be seen by the performance numbers below there is a significant speedup in the IntMap and LongMap benchmarks with this change.

IntMap	Bit ops as foreign functions	Bit ops as primitives	% change
unionBenchmark	13906	10033	27.85%
intersectionBenchmark	10281	6501	36.77%
lookupBenchmark	1203	704	41.48%
filterBenchmark	5796	3969	31.52%
partitionBenchmark	8969	7126	20.55%
deleteBenchmark	14469	10390	28.19%

fromListBenchmark	5640	3703	34.34%
fromAscListBenchmark	5625	3704	34.15%
fromDistinctAscListBenchmark	5610	3719	33.71%

LongMap			
unionBenchmark	15250	11625	23.77%
intersectionBenchmark	11547	8141	29.50%
lookupBenchmark	1407	938	33.33%
filterBenchmark	6500	4829	25.71%
partitionBenchmark	9516	7969	16.26%
deleteBenchmark	15843	11375	28.20%
fromListBenchmark	6125	4594	25.00%
fromAscListBenchmark	6125	4609	24.75%
fromDistinctAscListBenchmark	6109	4594	24.80%

December 7, 2005: Lifted let variable definitions.

This change is the result of a problem encountered when compiling some automatically generated CAL code. The CAL code in question declared over 100 let-variables in a function body. Each of these let-variables had a very large definition. As a result the generated java code exceeded the limits of the Java class file format. In this case the limit exceeded was that no single method in a class can have more than 64Kb of byte-code.

The solution to this problem was to lift the definition of each non-recursive let-variable into its own java method. Thus the generated byte-code was distributed across multiple methods. This did, however, have some implications for the existing set of optimizations.

Prior to this change the definition of a single-reference non-recursive let-variable was in-lined at the point of reference. This allowed the generated code for the definition to be more optimal based on the usage of the let-variable (i.e. strict, lazy, or unboxed). With the lifting of let-variable definitions this is no longer possible. The best that can be done is to in-line the call to the lifted function and avoid the creation of a local variable in the java code. The code generation is still trying to be smart about how the definition of a single-reference let-variable is compiled however it is not always possible to determine this without more analysis than is currently being done.

These benchmarks test various types/uses of single-reference let-variables. Each benchmark has two versions. One version is written using a single-reference let variable and the other version is written with the let variable definition manually in-lined. This is intended to show the difference in performance between the previously generated code which used direct in-lining and the newer code which generates lifted let-variable definitions.

These benchmarks are in the Benchmarks.cal module.

For most of the benchmarks the difference is minimal, i.e. < 3%. However two cases are actually faster by 3.4% and 6.5%.
The biggest difference is in benchmark number 7. This is a case where a let variable is used strictly, but the definition function builds a lazy graph because we cannot determine that it is safe to compile the definition strictly. Also it is a case where there are two let variables, both are single-reference, so the overhead of calling

the definition functions is doubled, compared to the in-lined case. This indicates the importance of compiling the definition of a let variable strictly if it as always going to be used strictly.

	in-lined let var	single-ref. let var	% change
letVarBenchmark_1	10532	10765	-2.21%
letVarBenchmark_2	703	703	0.00%
letVarBenchmark_3	16218	15671	3.37%
letVarBenchmark_4	2031	2078	-2.31%
letVarBenchmark_5	125	125	0.00%
letVarBenchmark_6	4109	3843	6.47%
letVarBenchmark_7	625	2906	-364.96%

March 24, 2006: LECC runtime enhancements.

Refactoring of some existing runtime classes :
RTApp was renamed to RTFullApp and is the base class for any nodes representing a fully saturated application.
Generated, SC specific, fully saturated application node classes (i.e. RTAppL, RTAppS) now derive from RTFullApp.
A static inner class RTFullApp.General was introduced. This is the base class for all full application nodes not tied to a specific supercombinator. Static inner classes of general were created (_1, _2, ..., _15) to represent fully saturated applications of SCs up to arity 15. Each of these inner classes has static inner classes L and S for cases where we can't guarantee strict arguments are in whnf (L) and when we can guarantee that strict arguments are in WHNF (S). So an instance of RTFullApp.General._3.S is a fully saturated application of an arity 3 supercombinator where the strict arguments are guaranteed to already be in WHNF, and replaces the previously existing class RTApp3S.
This refactoring allowed a lot of code that was common between classes like RTApp3L and RTApp3S to be put into the the common base class _3.

Added new node classes to represent partial applications of up to 15 arguments. These classes are derived from RTSupercombinator. They behave like an RTSupercombinator of arity N where N is the arity of the originally applied supercombinator less the number of provided arguments. The classes then override the appropriate f methods to complete the application and invoke the original supercombinator. This cuts down on the number of general application nodes created. Previously a partial app of 3 arguments would create three general application nodes. Now a single instance of an RTPartialApp derived class is created. Also when the full application chain is built traversing the graph to do the reduction will take less time because there are fewer nodes.

The meaning the the getArity() and getArgCount() methods have been changed. Previously invoking these methods on a graph node would return the arity/arg count for the entire graph of which the node was the root. Now they return the value only for the node on which the method is invoked. This change was made to cut down on the number of graph traversals when doing reductions. It makes more sense to traverse the graph once, examining each node, than to call getter functions for each value when each getter function will do its own traversal of the graph.

As part of this change RTFullApp implements final versions of getArity()/getArgCount() that simply return zero. This is valid since before reduction an RTFullApp instance has exactly enough arguments for its supercombinator and after reduction an RTFullApp instance is simply an indirection to its result. This means that we don't have to generate these functions in the generation application node classes.

Removed the getSupercombinator() function. Ultimately all getSupercombinator() did was traverse to the left hand side of the graph and get the RTSupercombinator instance there. The f method was then invoked on this supercombinator. Since we are traversing the graph to the left hand side via the general traversal mechanism and the f method is actually a method of RTValue we can just invoke the f method on the graph node at the left of the graph. This saves a traversal over the graph when reducing.

Changed the base implementations of the fnL methods to simply build an application chain of the provided arguments and then return. This allows the code generation to generate calls to the fnL functions when dealing with an application of a local value in a strict context. For example if fn was an argument of a functional type an application of fn in a strict context would generate code like:
'fn.apply(arg1).apply(arg2).apply(arg3).evaluate()'. Now we generate 'fn.f3L(arg1, arg2, arg3).evaluate()'. If fn is in actuality a supercombinator instance or an instance of a partial application node the call the fnL will execute the function logic, saving the building of an application graph that will immediately be discarded as it is evaluated. Otherwise the fnL method will simply build the application chain producing the previous behaviour. This change can be quite beneficial. For example, in one of the EFashion benchmarks the number of application nodes allocated was reduced by 12.6 million, or ~6%.

Inlined the assignment of the local case variable. When generating a switch in java from a CAL case we would generate something like:
RTValue $case1 = case expression;
switch($case1.getOrdinal())
Examination of generated bytecode has shown that it is slightly more efficient to generate:
RTValue $case1;
switch(($case1 = case expression).getOrdinal())

Changed the reduction of general application nodes. When reducing a general application node three things need to be extracted from the graph: arity, arg count, and the supercombinator. Previously the arity and arg count were retrieved by calling getArity() and getArgCount() on the graph root. getArity() would traverse to the left hand of the graph and get the arity of the left-most node. getArgCount would traverse the graph recursively accumulating the arg count for the whole graph. Similarly getSupercombinator() would traverse the graph to retrieve the supercombinator on the left hand side. Thus getting these three values involved traversing the graph three. The previously mentioned changes to getArgCount() and getArity() allow the reduce method to travers the graph to the left-most node via a simple loop. Each node in the traversal is queried for its arg count to accumulate the total arg count. Once the left most node is reached it can be queried for the arity. As previously noted the getSupercombinator() function has been deleted since the left-most node is the one we want to invoke the f method on. With these changes we only need to traverse the graph once, instead of three times.

Changed the runtime parameter checks in RT... classes to use assert.

Benchmarking results:


```
Benchmarks: Benchmarks.badRegressionGemTest 500
   Base benchmarks avg: 5192, med: 5188, stdDev: 9.0, nRuns: 3
   New Benchmarks avg: 5495, med: 5485, stdDev: 18.19, nRuns: 3        %change: -5.84  running %change: -5.84

Benchmarks: Benchmarks.getNthPrime 20000
   Base benchmarks avg: 8297, med: 8297, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 8156, med: 8156, stdDev: 0.0, nRuns: 3  %change: 1.7    running %change: 1.7

Benchmarks: BonusRatings.bonusRatedPeople (BonusRatings.peopleScoresN 800)
   Base benchmarks avg: 15968, med: 16000, stdDev: 68.24, nRuns: 3
   New Benchmarks avg: 15864, med: 15859, stdDev: 23.89, nRuns: 3      %change: 0.65   running %change: 0.65

Benchmarks: applyTest 1.0 2.0 60000000
   Base benchmarks avg: 13234, med: 13234, stdDev: 16.0, nRuns: 3
   New Benchmarks avg: 9734, med: 9734, stdDev: 0.0, nRuns: 3  %change: 26.45  running %change: 26.45

Benchmarks: benchmarkAverage (expandedDoubleList 2000000)
   Base benchmarks avg: 3796, med: 3797, stdDev: 15.52, nRuns: 3
   New Benchmarks avg: 3119, med: 3125, stdDev: 23.47, nRuns: 3        %change: 17.83  running %change: 17.83

Benchmarks: benchmarkAverage (upFromTo 1.0 2000000.0)
   Base benchmarks avg: 2557, med: 2562, stdDev: 8.94, nRuns: 3
   New Benchmarks avg: 1895, med: 1891, stdDev: 9.0, nRuns: 3  %change: 25.89  running %change: 25.89

Benchmarks: benchmarkPartialApplications 200000.0
   Base benchmarks avg: 22301, med: 22297, stdDev: 9.0, nRuns: 3
   New Benchmarks avg: 16713, med: 16812, stdDev: 227.64, nRuns: 3     %change: 25.06  running %change: 25.06

Benchmarks: buildIntBoxTree 12
   Base benchmarks avg: 1593, med: 1594, stdDev: 1.0, nRuns: 3
   New Benchmarks avg: 1281, med: 1250, stdDev: 54.26, nRuns: 3        %change: 19.59  running %change: 19.59

Benchmarks: dynamicMapLookupBenchmark 1000000
   Base benchmarks avg: 21884, med: 21890, stdDev: 70.15, nRuns: 3
   New Benchmarks avg: 18635, med: 18640, stdDev: 23.89, nRuns: 3      %change: 14.85  running %change: 14.85

Benchmarks: last (take 6000000 (upFrom 1.0))
   Base benchmarks avg: 3770, med: 3781, stdDev: 32.75, nRuns: 3
   New Benchmarks avg: 3677, med: 3687, stdDev: 47.79, nRuns: 3        %change: 2.47   running %change: 2.47

Benchmarks: letTestHelper 10000000
   Base benchmarks avg: 5047, med: 5047, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 4630, med: 4625, stdDev: 9.21, nRuns: 3 %change: 8.26   running %change: 8.26

Benchmarks: letVarBenchmark_1_a 30000000
   Base benchmarks avg: 9796, med: 9765, stdDev: 82.58, nRuns: 3
   New Benchmarks avg: 9932, med: 9875, stdDev: 127.44, nRuns: 3       %change: -1.39  running %change: -1.39

Benchmarks: letVarBenchmark_1_b 30000000
   Base benchmarks avg: 9682, med: 9625, stdDev: 156.1, nRuns: 3
   New Benchmarks avg: 10083, med: 10047, stdDev: 213.29, nRuns: 3     %change: -4.14  running %change: -4.14

Benchmarks: letVarBenchmark_2_a 30000000
   Base benchmarks avg: 693, med: 688, stdDev: 8.66, nRuns: 3
   New Benchmarks avg: 703, med: 703, stdDev: 0.0, nRuns: 3    %change: -1.44  running %change: -1.44

Benchmarks: letVarBenchmark_2_b 30000000
   Base benchmarks avg: 693, med: 688, stdDev: 8.66, nRuns: 3
   New Benchmarks avg: 703, med: 703, stdDev: 0.0, nRuns: 3    %change: -1.44  running %change: -1.44

Benchmarks: letVarBenchmark_3_a 30000000
   Base benchmarks avg: 15192, med: 15202, stdDev: 17.32, nRuns: 3
   New Benchmarks avg: 15286, med: 15312, stdDev: 59.43, nRuns: 3      %change: -0.62  running %change: -0.62

Benchmarks: letVarBenchmark_3_b 30000000
   Base benchmarks avg: 14905, med: 14906, stdDev: 15.52, nRuns: 3
   New Benchmarks avg: 14848, med: 14859, stdDev: 32.34, nRuns: 3      %change: 0.38   running %change: 0.38

Benchmarks: letVarBenchmark_4_a 30000000
   Base benchmarks avg: 2083, med: 2078, stdDev: 8.66, nRuns: 3
   New Benchmarks avg: 2072, med: 2078, stdDev: 9.27, nRuns: 3 %change: 0.53   running %change: 0.53

Benchmarks: letVarBenchmark_4_b 30000000
   Base benchmarks avg: 2031, med: 2031, stdDev: 15.49, nRuns: 3
   New Benchmarks avg: 2036, med: 2032, stdDev: 8.36, nRuns: 3 %change: -0.25  running %change: -0.25

Benchmarks: letVarBenchmark_5_a 30000000
   Base benchmarks avg: 125, med: 125, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 125, med: 125, stdDev: 0.0, nRuns: 3    %change: 0.0    running %change: 0.0
```



```
Benchmarks: letVarBenchmark_5_b 30000000
   Base benchmarks avg: 125, med: 125, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 125, med: 125, stdDev: 0.0, nRuns: 3    %change: 0.0    running %change: 0.0

Benchmarks: letVarBenchmark_6_a 30000000
   Base benchmarks avg: 3468, med: 3469, stdDev: 1.0, nRuns: 3
   New Benchmarks avg: 3646, med: 3641, stdDev: 8.66, nRuns: 3 %change: -5.13  running %change: -5.13

Benchmarks: letVarBenchmark_6_b 30000000
   Base benchmarks avg: 3453, med: 3453, stdDev: 16.0, nRuns: 3
   New Benchmarks avg: 3625, med: 3625, stdDev: 15.0, nRuns: 3 %change: -4.98  running %change: -4.98

Benchmarks: letVarBenchmark_7_a 30000000
   Base benchmarks avg: 661, med: 656, stdDev: 9.21, nRuns: 3
   New Benchmarks avg: 666, med: 671, stdDev: 8.42, nRuns: 3   %change: -0.76  running %change: -0.76

Benchmarks: letVarBenchmark_7_b 30000000
   Base benchmarks avg: 625, med: 625, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 614, med: 610, stdDev: 9.0, nRuns: 3    %change: 1.76   running %change: 1.76

Benchmarks: letVarScopingBenchmark_1 3000000
   Base benchmarks avg: 9031, med: 9031, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 2927, med: 2922, stdDev: 23.89, nRuns: 3        %change: 67.59  running %change: 67.59

Benchmarks: letVarScopingBenchmark_2_a 9000000
   Base benchmarks avg: 3234, med: 3219, stdDev: 27.14, nRuns: 3
   New Benchmarks avg: 3192, med: 3219, stdDev: 45.61, nRuns: 3        %change: 1.3    running %change: 1.3

Benchmarks: letVarScopingBenchmark_2_b 9000000
   Base benchmarks avg: 3224, med: 3219, stdDev: 23.89, nRuns: 3
   New Benchmarks avg: 3203, med: 3187, stdDev: 41.38, nRuns: 3        %change: 0.65   running %change: 0.65

Benchmarks: letVarScopingBenchmark_3_a 9000000
   Base benchmarks avg: 3474, med: 3484, stdDev: 47.79, nRuns: 3
   New Benchmarks avg: 3411, med: 3391, stdDev: 50.29, nRuns: 3        %change: 1.81   running %change: 1.81

Benchmarks: letVarScopingBenchmark_3_b 9000000
   Base benchmarks avg: 3494, med: 3484, stdDev: 32.35, nRuns: 3
   New Benchmarks avg: 3437, med: 3453, stdDev: 40.81, nRuns: 3        %change: 1.63   running %change: 1.63

Benchmarks: primitiveOpOverhead2Helper 60000000
   Base benchmarks avg: 609, med: 610, stdDev: 1.0, nRuns: 3

Benchmarks: sumIntBoxes (incrementIntBoxes (buildIntBoxTree 14))
   Base benchmarks avg: 7599, med: 7672, stdDev: 197.87, nRuns: 3
   New Benchmarks avg: 7281, med: 7250, stdDev: 53.69, nRuns: 3        %change: 4.18   running %change: 4.18

EaselBenchmarks: scorecardBenchmark
   Base benchmarks avg: 546, med: 547, stdDev: 15.52, nRuns: 3
   New Benchmarks avg: 531, med: 515, stdDev: 41.95, nRuns: 3  %change: 2.75   running %change: 2.75

EaselBenchmarks: workbookBenchmark4 10000
   Base benchmarks avg: 9489, med: 9484, stdDev: 24.43, nRuns: 3
   New Benchmarks avg: 9218, med: 9218, stdDev: 0.0, nRuns: 3  %change: 2.86   running %change: 2.86

ISL_DemoScript: benchmarkPerfMgtData
   Base benchmarks avg: 137308, med: 137433, stdDev: 258.28, nRuns: 3
   New Benchmarks avg: 118835, med: 117794, stdDev: 1885.63, nRuns: 3  %change: 13.45  running %change: 13.45

IntMap: deleteBenchmark 1
   Base benchmarks avg: 10291, med: 10281, stdDev: 32.74, nRuns: 3
   New Benchmarks avg: 9333, med: 9328, stdDev: 8.66, nRuns: 3 %change: 9.31   running %change: 9.31

IntMap: filterBenchmark 1
   Base benchmarks avg: 3890, med: 3890, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 3542, med: 3547, stdDev: 8.66, nRuns: 3 %change: 8.95   running %change: 8.95

IntMap: fromAscListBenchmark 1
   Base benchmarks avg: 3838, med: 3828, stdDev: 17.88, nRuns: 3
   New Benchmarks avg: 3328, med: 3328, stdDev: 0.0, nRuns: 3  %change: 13.29  running %change: 13.29

IntMap: fromDistinctAscListBenchmark 1
   Base benchmarks avg: 3828, med: 3828, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 3313, med: 3313, stdDev: 0.0, nRuns: 3  %change: 13.45  running %change: 13.45

IntMap: fromListBenchmark 1
   Base benchmarks avg: 3828, med: 3828, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 3317, med: 3313, stdDev: 9.0, nRuns: 3  %change: 13.35  running %change: 13.35
```

```
IntMap: intersectionBenchmark 1
   Base benchmarks avg: 6458, med: 6453, stdDev: 23.45, nRuns: 3
   New Benchmarks avg: 5994, med: 6000, stdDev: 9.27, nRuns: 3 %change: 7.18   running %change: 7.18

IntMap: lookupBenchmark 1
   Base benchmarks avg: 593, med: 593, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 541, med: 546, stdDev: 8.42, nRuns: 3   %change: 8.77   running %change: 8.77

IntMap: partitionBenchmark 1
   Base benchmarks avg: 6416, med: 6422, stdDev: 9.27, nRuns: 3
   New Benchmarks avg: 6000, med: 6000, stdDev: 0.0, nRuns: 3  %change: 6.48   running %change: 6.48

IntMap: unionBenchmark 1
   Base benchmarks avg: 8931, med: 8921, stdDev: 18.49, nRuns: 3
   New Benchmarks avg: 8260, med: 8265, stdDev: 8.66, nRuns: 3 %change: 7.51   running %change: 7.51

M2: M2.mainM2
   Base benchmarks avg: 2005, med: 2031, stdDev: 119.58, nRuns: 3
   New Benchmarks avg: 1739, med: 1687, stdDev: 177.79, nRuns: 3       %change: 13.27  running %change: 13.27

Nofib: List.subscript digits_of_e1 800
   Base benchmarks avg: 8989, med: 8952, stdDev: 91.3, nRuns: 3
   New Benchmarks avg: 8786, med: 8781, stdDev: 9.21, nRuns: 3 %change: 2.26   running %change: 2.26

Nofib: exp3 8
   Base benchmarks avg: 2531, med: 2531, stdDev: 0.0, nRuns: 3
   New Benchmarks avg: 2562, med: 2563, stdDev: 1.0, nRuns: 3  %change: -1.22  running %change: -1.22

SMEBenchmarkTestsEFashion: testReports Dense MinLevels
   Base benchmarks avg: 12067, med: 12078, stdDev: 109.91, nRuns: 3
   New Benchmarks avg: 11494, med: 11469, stdDev: 44.74, nRuns: 3      %change: 4.75   running %change: 4.75

SMEBenchmarkTestsEFashion: testReports Sparse AllLevels
   Base benchmarks avg: 216659, med: 216727, stdDev: 1071.58, nRuns: 3
   New Benchmarks avg: 197766, med: 197855, stdDev: 167.31, nRuns: 3   %change: 8.72   running %change: 8.72

SimpleMolap2: verifySimpleCube1
   Base benchmarks avg: 5182, med: 5172, stdDev: 32.74, nRuns: 3
   New Benchmarks avg: 3354, med: 3344, stdDev: 18.19, nRuns: 3        %change: 35.28  running %change: 35.28
```

Some of the benchmarks showed a decrease in performance of 4-5%. The generated code for these benchmarks from before/after the changes was examined and no significant differences were found. Re-running the benchmarks in question produced comparable results to the previous build.

April 6, 2006: Unboxed return value optimization.

When generating code for a CAL function in lecc we now are generating function bodies which can return an unboxed value directly. When the result type of a CAL function can be unboxed we are generating the regular function body. We then do a copy/transformation on the function body to return unboxed values.
When generating code for an expression which resolves to an unboxed value we are now handling the case where the expression is a fully saturated function application by generating a direct call to the 'fUnboxed' method of the generated function class.

Benchmark results were obtained using:
P4 3GHz with 2Gbyte RAM.
OS - Windows Server 2003
JDK 1.4.2_06, client JVM, jvm command line arguments: -Xms512M -Xmx512M -Xss1M

```
Benchmarks: Benchmarks.badRegressionGemTest 500
    boxed returns avg: 4636, med: 4624, stdDev: 28.6 (0.62%), stdError: 9.53 (0.21%),
nRuns: 9
    unboxed returns avg: 4857, med: 4863, stdDev: 16.52 (0.34%), stdError: 5.51 (0.11%),
nRuns: 9    %change: -4.77    running %change: -4.77
```



```
 Benchmarks: Benchmarks.getNthPrime 20000
    boxed returns avg: 6822, med: 6829, stdDev: 8.12 (0.12%), stdError: 2.71 (0.04%),
nRuns: 9
    unboxed returns avg: 7005, med: 6988, stdDev: 53.21 (0.76%), stdError: 17.74 (0.25%),
nRuns: 9    %change: -2.68    running %change: -2.68

 Benchmarks: BonusRatings.bonusRatedPeople (BonusRatings.peopleScoresN 800)
    boxed returns avg: 14235, med: 14241, stdDev: 177.7 (1.25%), stdError: 59.23 (0.42%),
nRuns: 9
    unboxed returns avg: 14192, med: 14202, stdDev: 89.05 (0.63%), stdError: 29.68
(0.21%), nRuns: 9 %change: 0.3      running %change: 0.3

 Benchmarks: applyTest 1.0 2.0 60000000
    boxed returns avg: 12200, med: 12202, stdDev: 14.35 (0.12%), stdError: 4.78 (0.04%),
nRuns: 9
    unboxed returns avg: 12268, med: 12264, stdDev: 27.03 (0.22%), stdError: 9.01
(0.07%), nRuns: 9 %change: -0.56    running %change: -0.56

 Benchmarks: benchmarkAverage (expandedDoubleList 2000000)
    boxed returns avg: 3537, med: 3058, stdDev: 628.31 (17.76%), stdError: 209.44
(5.92%), nRuns: 9
    unboxed returns avg: 3527, med: 3042, stdDev: 637.26 (18.07%), stdError: 212.42
(6.02%), nRuns: 9 %change: 0.28     running %change: 0.28

 Benchmarks: benchmarkAverage (upFromTo 1.0 2000000.0)
    boxed returns avg: 1747, med: 1754, stdDev: 8.36 (0.48%), stdError: 2.79 (0.16%),
nRuns: 9
    unboxed returns avg: 1757, med: 1763, stdDev: 10.86 (0.62%), stdError: 3.62 (0.21%),
nRuns: 9    %change: -0.57    running %change: -0.57

 Benchmarks: benchmarkPartialApplications 200000.0
    boxed returns avg: 15408, med: 15408, stdDev: 19.44 (0.13%), stdError: 6.48 (0.04%),
nRuns: 9
    unboxed returns avg: 15410, med: 15400, stdDev: 23.19 (0.15%), stdError: 7.73
(0.05%), nRuns: 9 %change: -0.01    running %change: -0.01

 Benchmarks: buildIntBoxTree 12
    boxed returns avg: 1383, med: 1323, stdDev: 216.22 (15.63%), stdError: 72.07 (5.21%),
nRuns: 9
    unboxed returns avg: 1404, med: 1315, stdDev: 267.72 (19.07%), stdError: 89.24
(6.36%), nRuns: 9 %change: -1.52    running %change: -1.52

 Benchmarks: dynamicMapLookupBenchmark 1000000
    boxed returns avg: 16115, med: 16109, stdDev: 23.55 (0.15%), stdError: 8.33 (0.05%),
nRuns: 8
    unboxed returns avg: 15880, med: 15883, stdDev: 28.77 (0.18%), stdError: 9.59
(0.06%), nRuns: 9 %change: 1.46     running %change: 1.46

 Benchmarks: last (take 6000000 (upFrom 1.0))
    boxed returns avg: 3408, med: 3407, stdDev: 12.08 (0.35%), stdError: 4.03 (0.12%),
nRuns: 9
    unboxed returns avg: 3458, med: 3455, stdDev: 10.29 (0.3%), stdError: 3.43 (0.1%),
nRuns: 9    %change: -1.47    running %change: -1.47

 Benchmarks: letTestHelper 10000000
    boxed returns avg: 4628, med: 4626, stdDev: 5.19 (0.11%), stdError: 1.73 (0.04%),
nRuns: 9
```



```
    unboxed returns avg: 4513, med: 4482, stdDev: 79.37 (1.76%), stdError: 26.46 (0.59%),
nRuns: 9    %change: 2.48     running %change: 2.48

 Benchmarks: letVarBenchmark_1_a 30000000
    boxed returns avg: 8184, med: 8179, stdDev: 11.78 (0.14%), stdError: 3.93 (0.05%),
nRuns: 9
    unboxed returns avg: 8209, med: 8214, stdDev: 7.81 (0.1%), stdError: 2.6 (0.03%),
nRuns: 9    %change: -0.31    running %change: -0.31

 Benchmarks: letVarBenchmark_1_b 30000000
    boxed returns avg: 8069, med: 8054, stdDev: 42.62 (0.53%), stdError: 14.21 (0.18%),
nRuns: 9
    unboxed returns avg: 8110, med: 8076, stdDev: 93.12 (1.15%), stdError: 31.04 (0.38%),
nRuns: 9    %change: -0.51    running %change: -0.51

 Benchmarks: letVarBenchmark_2_a 30000000
    boxed returns avg: 422, med: 421, stdDev: 5.47 (1.3%), stdError: 1.82 (0.43%), nRuns:
9
    unboxed returns avg: 449, med: 437, stdDev: 23.19 (5.16%), stdError: 7.73 (1.72%),
nRuns: 9    %change: -6.4     running %change: -6.4

 Benchmarks: letVarBenchmark_2_b 30000000
    boxed returns avg: 422, med: 421, stdDev: 5.09 (1.21%), stdError: 1.7 (0.4%), nRuns:
9
    unboxed returns avg: 425, med: 422, stdDev: 6.92 (1.63%), stdError: 2.31 (0.54%),
nRuns: 9    %change: -0.71    running %change: -0.71

 Benchmarks: letVarBenchmark_3_a 30000000
    boxed returns avg: 13187, med: 13179, stdDev: 23.45 (0.18%), stdError: 7.82 (0.06%),
nRuns: 9
    unboxed returns avg: 13235, med: 13232, stdDev: 34.45 (0.26%), stdError: 11.48
(0.09%), nRuns: 9 %change: -0.36    running %change: -0.36

 Benchmarks: letVarBenchmark_3_b 30000000
    boxed returns avg: 13317, med: 13321, stdDev: 7.48 (0.06%), stdError: 2.49 (0.02%),
nRuns: 9
    unboxed returns avg: 13397, med: 13389, stdDev: 20.92 (0.16%), stdError: 6.97
(0.05%), nRuns: 9 %change: -0.6     running %change: -0.6

 Benchmarks: letVarBenchmark_4_a 30000000
    boxed returns avg: 1802, med: 1807, stdDev: 7.74 (0.43%), stdError: 2.58 (0.14%),
nRuns: 9
    unboxed returns avg: 1298, med: 1297, stdDev: 5.47 (0.42%), stdError: 1.82 (0.14%),
nRuns: 9    %change: 27.97    running %change: 27.97

 Benchmarks: letVarBenchmark_4_b 30000000
    boxed returns avg: 1741, med: 1745, stdDev: 7.07 (0.41%), stdError: 2.36 (0.14%),
nRuns: 9
    unboxed returns avg: 1270, med: 1266, stdDev: 8.0 (0.63%), stdError: 2.67 (0.21%),
nRuns: 9    %change: 27.05    running %change: 27.05

 Benchmarks: letVarBenchmark_5_a 30000000
    boxed returns avg: 78, med: 78, stdDev: 0.0 (0.0%), stdError: 0.0 (0.0%), nRuns: 9
    unboxed returns avg: 69, med: 70, stdDev: 8.12 (11.77%), stdError: 2.71 (3.93%),
nRuns: 9    %change: 11.54    running %change: 11.54

 Benchmarks: letVarBenchmark_5_b 30000000
```



```
    boxed returns avg: 77, med: 78, stdDev: 7.74 (10.05%), stdError: 2.58 (3.35%), nRuns:
9
    unboxed returns avg: 78, med: 78, stdDev: 0.0 (0.0%), stdError: 0.0 (0.0%), nRuns: 9
      %change: -1.3     running %change: -1.3

 Benchmarks: letVarBenchmark_6_a 30000000
    boxed returns avg: 3011, med: 3008, stdDev: 10.44 (0.35%), stdError: 3.48 (0.12%),
nRuns: 9
    unboxed returns avg: 2961, med: 2968, stdDev: 8.36 (0.28%), stdError: 2.79 (0.09%),
nRuns: 9    %change: 1.66     running %change: 1.66

 Benchmarks: letVarBenchmark_6_b 30000000
    boxed returns avg: 3035, med: 3038, stdDev: 6.85 (0.23%), stdError: 2.28 (0.08%),
nRuns: 9
    unboxed returns avg: 3003, med: 3000, stdDev: 6.85 (0.23%), stdError: 2.28 (0.08%),
nRuns: 9    %change: 1.05     running %change: 1.05

 Benchmarks: letVarBenchmark_7_a 30000000
    boxed returns avg: 528, med: 530, stdDev: 5.0 (0.95%), stdError: 1.67 (0.32%), nRuns:
9
    unboxed returns avg: 236, med: 234, stdDev: 5.19 (2.2%), stdError: 1.73 (0.73%),
nRuns: 9    %change: 55.3     running %change: 55.3

 Benchmarks: letVarBenchmark_7_b 30000000
    boxed returns avg: 483, med: 483, stdDev: 7.74 (1.6%), stdError: 2.58 (0.53%), nRuns:
9
    unboxed returns avg: 234, med: 234, stdDev: 0.0 (0.0%), stdError: 0.0 (0.0%), nRuns:
9     %change: 51.55    running %change: 51.55

 Benchmarks: letVarScopingBenchmark_1 3000000
    boxed returns avg: 2711, med: 2712, stdDev: 7.74 (0.29%), stdError: 2.58 (0.1%),
nRuns: 9
    unboxed returns avg: 2617, med: 2624, stdDev: 8.36 (0.32%), stdError: 2.79 (0.11%),
nRuns: 9    %change: 3.47     running %change: 3.47

 Benchmarks: letVarScopingBenchmark_2_a 9000000
    boxed returns avg: 2747, med: 2743, stdDev: 11.09 (0.4%), stdError: 3.7 (0.13%),
nRuns: 9
    unboxed returns avg: 2626, med: 2625, stdDev: 5.0 (0.19%), stdError: 1.67 (0.06%),
nRuns: 9    %change: 4.4      running %change: 4.4

 Benchmarks: letVarScopingBenchmark_2_b 9000000
    boxed returns avg: 2841, med: 2836, stdDev: 7.68 (0.27%), stdError: 2.56 (0.09%),
nRuns: 9
    unboxed returns avg: 2628, med: 2625, stdDev: 6.7 (0.25%), stdError: 2.23 (0.08%),
nRuns: 9    %change: 7.5      running %change: 7.5

 Benchmarks: letVarScopingBenchmark_3_a 9000000
    boxed returns avg: 2985, med: 2984, stdDev: 8.24 (0.28%), stdError: 2.75 (0.09%),
nRuns: 9
    unboxed returns avg: 2897, med: 2890, stdDev: 8.42 (0.29%), stdError: 2.81 (0.1%),
nRuns: 9    %change: 2.95     running %change: 2.95

 Benchmarks: letVarScopingBenchmark_3_b 9000000
    boxed returns avg: 2987, med: 2992, stdDev: 8.0 (0.27%), stdError: 2.67 (0.09%),
nRuns: 9
```

```
    Unboxed returns avg: 2888, med: 2890, stdDev: 9.32 (0.32%), stdError: 3.11 (0.11%),
nRuns: 9    %change: 3.31     running %change: 3.31

 Benchmarks: sumIntBoxes (incrementIntBoxes (buildIntBoxTree 14))
    boxed returns avg: 6626, med: 6628, stdDev: 19.67 (0.3%), stdError: 6.56 (0.1%),
nRuns: 9
    unboxed returns avg: 6885, med: 6885, stdDev: 23.02 (0.33%), stdError: 7.67 (0.11%),
nRuns: 9    %change: -3.91    running %change: -3.91

 EFashionReportBenchmark: test1
    boxed returns avg: 7137, med: 7152, stdDev: 42.21 (0.59%), stdError: 14.07 (0.2%),
nRuns: 9
    unboxed returns avg: 6984, med: 6985, stdDev: 63.54 (0.91%), stdError: 21.18 (0.3%),
nRuns: 9    %change: 2.14     running %change: 2.14

 EFashionReportBenchmark: test2
    boxed returns avg: 9370, med: 9251, stdDev: 408.13 (4.36%), stdError: 136.04 (1.45%),
nRuns: 9
    unboxed returns avg: 9257, med: 9164, stdDev: 395.25 (4.27%), stdError: 131.75
(1.42%), nRuns: 9 %change: 1.21     running %change: 1.21

 EFashionReportBenchmark: test3
    boxed returns avg: 9349, med: 9228, stdDev: 384.72 (4.12%), stdError: 128.24 (1.37%),
nRuns: 9
    unboxed returns avg: 9230, med: 9109, stdDev: 353.94 (3.83%), stdError: 117.98
(1.28%), nRuns: 9 %change: 1.27     running %change: 1.27

 EaselBenchmarks: scorecardBenchmark
    boxed returns avg: 512, med: 530, stdDev: 54.25 (10.6%), stdError: 18.08 (3.53%),
nRuns: 9
    unboxed returns avg: 512, med: 531, stdDev: 50.39 (9.84%), stdError: 16.8 (3.28%),
nRuns: 9    %change: 0.0      running %change: 0.0

 EaselBenchmarks: workbookBenchmark4 10000
    boxed returns avg: 9145, med: 9141, stdDev: 32.64 (0.36%), stdError: 10.88 (0.12%),
nRuns: 9
    unboxed returns avg: 9357, med: 9359, stdDev: 29.81 (0.32%), stdError: 9.94 (0.11%),
nRuns: 9    %change: -2.32    running %change: -2.32

 ISL_DemoScript: benchmarkPerfMgtData
    boxed returns avg: 110199, med: 110348, stdDev: 509.89 (0.46%), stdError: 169.96
(0.15%), nRuns: 9
    unboxed returns avg: 110584, med: 110538, stdDev: 537.2 (0.49%), stdError: 179.07
(0.16%), nRuns: 9 %change: -0.35    running %change: -0.35

 M2: M2.mainM2
    boxed returns avg: 1503, med: 1490, stdDev: 167.31 (11.13%), stdError: 55.77 (3.71%),
nRuns: 9
    unboxed returns avg: 1529, med: 1523, stdDev: 139.89 (9.15%), stdError: 46.63
(3.05%), nRuns: 9 %change: -1.73    running %change: -1.73

 M2: intArray_javaQuicksort (randomIntArray 2000000)
    boxed returns avg: 4708, med: 4948, stdDev: 922.36 (19.59%), stdError: 307.45
(6.53%), nRuns: 9
    unboxed returns avg: 4656, med: 4913, stdDev: 901.5 (19.36%), stdError: 300.5
(6.45%), nRuns: 9 %change: 1.1      running %change: 1.1
```

```
 M2: java_sieveBasedGetNthPrime 300000
    boxed returns avg: 1418, med: 1418, stdDev: 0.0 (0.0%), stdError: 0.0 (0.0%), nRuns:
9
    unboxed returns avg: 1421, med: 1422, stdDev: 0.0 (0.0%), stdError: 0.0 (0.0%),
nRuns: 9    %change: -0.21    running %change: -0.21

 M2: quicksortIntArray_v1 (randomIntArray 2000000)
    boxed returns avg: 10203, med: 10036, stdDev: 700.78 (6.87%), stdError: 233.59
(2.29%), nRuns: 9
    unboxed returns avg: 9938, med: 9913, stdDev: 649.07 (6.53%), stdError: 216.36
(2.18%), nRuns: 9 %change: 2.6      running %change: 2.6

 M2: quicksortIntArray_v3 (randomIntArray 2000000)
    boxed returns avg: 4909, med: 4831, stdDev: 511.36 (10.42%), stdError: 170.45
(3.47%), nRuns: 9
    unboxed returns avg: 4333, med: 4336, stdDev: 789.86 (18.23%), stdError: 263.29
(6.08%), nRuns: 9 %change: 11.73    running %change: 11.73

 M2: sieveBasedGetNthPrime 300000
    boxed returns avg: 1555, med: 1558, stdDev: 6.85 (0.44%), stdError: 2.28 (0.15%),
nRuns: 9
    unboxed returns avg: 1538, med: 1539, stdDev: 8.06 (0.52%), stdError: 2.69 (0.17%),
nRuns: 9    %change: 1.09     running %change: 1.09

 M2: sieveBasedGetNthPrime2 300000
    boxed returns avg: 1885, med: 1886, stdDev: 1.0 (0.05%), stdError: 0.33 (0.02%),
nRuns: 9
    unboxed returns avg: 1888, med: 1891, stdDev: 5.29 (0.28%), stdError: 1.76 (0.09%),
nRuns: 9    %change: -0.16    running %change: -0.16

 Nofib: List.subscript digits_of_e1 800
    boxed returns avg: 9389, med: 9094, stdDev: 573.36 (6.11%), stdError: 191.12 (2.04%),
nRuns: 9
    unboxed returns avg: 8801, med: 8452, stdDev: 677.73 (7.7%), stdError: 225.91
(2.57%), nRuns: 9 %change: 6.26     running %change: 6.26

 Nofib: exp3 8
    boxed returns avg: 2493, med: 2494, stdDev: 8.42 (0.34%), stdError: 2.81 (0.11%),
nRuns: 9
    unboxed returns avg: 2463, med: 2468, stdDev: 7.74 (0.31%), stdError: 2.58 (0.1%),
nRuns: 9    %change: 1.2      running %change: 1.2

 SMEBenchmarkTestsEFashion: testBaseline Dense MinLevels
    boxed returns avg: 9507, med: 9500, stdDev: 80.22 (0.84%), stdError: 26.74 (0.28%),
nRuns: 9
    unboxed returns avg: 9380, med: 9375, stdDev: 57.75 (0.62%), stdError: 19.25 (0.21%),
nRuns: 9    %change: 1.34     running %change: 1.34

 SMEBenchmarkTestsEFashion: testBaseline Sparse MinLevels
    boxed returns avg: 9956, med: 9963, stdDev: 36.7 (0.37%), stdError: 12.23 (0.12%),
nRuns: 9
    unboxed returns avg: 9786, med: 9781, stdDev: 23.57 (0.24%), stdError: 7.86 (0.08%),
nRuns: 9    %change: 1.71     running %change: 1.71

 SMEBenchmarkTestsEFashion: testReports Dense MinLevels
    boxed returns avg: 11179, med: 11169, stdDev: 57.87 (0.52%), stdError: 19.29 (0.17%),
nRuns: 9
```

```
    unboxed returns avg: 11093, med: 11094, stdDev: 54.61 (0.49%), stdError: 18.2
(0.16%), nRuns: 9 %change: 0.77     running %change: 0.77

 SMEBenchmarkTestsEFashion: testReports Sparse MinLevels
    boxed returns avg: 19352, med: 19330, stdDev: 65.9 (0.34%), stdError: 21.97 (0.11%),
nRuns: 9
    unboxed returns avg: 19227, med: 19171, stdDev: 152.67 (0.79%), stdError: 50.89
(0.26%), nRuns: 9 %change: 0.65     running %change: 0.65

 SimpleMolap2: verifySimpleCube1
    boxed returns avg: 3538, med: 3412, stdDev: 390.32 (11.03%), stdError: 130.11
(3.68%), nRuns: 9
    unboxed returns avg: 3206, med: 3062, stdDev: 393.19 (12.26%), stdError: 131.06
(4.09%), nRuns: 9 %change: 9.38     running %change: 9.38
```

Revision History:

November 8, 2002:
Created a benchmark document describing various benchmarks, the methodology for using them, and some initial results.

Dec. 11, 2002:
Updated Benchmark Results with figures for new compilation schemes based on original Peyton-Jones text.

Jan. 27, 2003:
Updated benchmark suite to remove obsolete benchmarks and increase the running times of the benchmarks.
Updated benchmark results with figures for tail call optimizations and space leak fixes.

Jan. 30, 2003:
Updated benchmark results with figures for replacing the I_Cont instruction with a continuation stack.

Feb. 12th, 2003:
Changes to eliminate continuation instructions, optimize pushing global nodes, and speed up unwinding.

March 24th 2003:
Added description of new data structure manipulation benchmark.
Performance results from eliminating CodeOffset and reducing casting and polymorphic function calls.

April 16th, 2003:
Added a description of new benchmark 'boxManipulation2', to be used in testing unwind performance.
Added the results of adding the new basic type String.
Added the results of changing the unwind mechanism to fix a bug.

July 3, 2003:
Added the results of fixing compiler problems related to overloaded functions.
Added initial results for the lecc machine and listed some areas for performance optimizations.

July 11, 2003:

Added the results of experimenting with how the lecc generated code accesses the singleton instances of supercombinators.

July 16, 2003:
Added the results of changing lecc code generation to create a java package for each CAL module and create a top-level class for each supercombinator.

July 17, 2003:
Updated benchmark results to do to the use of java switches for CAL case statements and optimization of top-level conditional expressions in LECC.

July 22, 2003:
Optimized and/or to be a primitive operation in strict contexts and removed the use of lazy extractors in LECC.

August 15, 2003:
Optimized the boxing/un-boxing of values in primitive operations and foreign functions. Optimized the generations/use of let variables.
Benchmarked changes to the prelude.

September 11, 2003:
Benchmarked the initial integration of lecc into ICE vs. using the standalone test application.

September 18, 2003:
Benchmarking of lecc and g-machine in ICE with a comparison between the client and server jvm.

September 25, 2003:
Added the results of optimizations to let variables.

September 30, 2003:
Added the results of optimizations to perfectly saturated applications.
Added a description of a new micro-benchmark, 'applyTest4000000'.

October 3, 2003:
Added results of optimizing fully saturated applications in strict contexts.

October 17, 2003:
Added results of optimizing case statements and a bug fix.

October 22, 2003:
Added the results from optimizing to reduce the use of static factory methods.

November 5, 2003:
Updated with test results from class loader experiments, changes to benchmark timing, optimization of fully saturated strict applications, and optimization of longer application chains.

November 14, 2003:

Updated with test results from adding statistics generation and checking the interrupt status of the currently executing thread. Also added some static/runtime code statistics.

November 19, 2003:
Added results of optimizing the representation of oversaturated applications of type class methods for Equals, Ord, and Num.

November 21, 2003:
Added the results of adding halting capability via checking a static flag in the executor.

December 16, 2003:
Added the results of passing an execution context through the runtime.

May 14, 2004:
Updated with test results from differentiating between recursive and non-recursive let variables.

June 29, 2004:
Use of strictness annotations on supercombinators and data constructors.

July 8, 2004:
Inlining of let variables.

August 2, 2004:
Added test results from using unboxed values. Added results of optimization experiments on the getNthPrime benchmark.

August 4, 2004:
Added test results from optimizing cases on two data constructor data types and expanding the use of strict application nodes.

August 27, 2004:
Added benchmarking results after core CAL functions were re-written to take advantage of un-boxing and strictness optimizations.

September 1, 2004:
Added benchmarking results after tail recursive functions were optimized to be loops. Added descriptions for the list traversal benchmarks.

September 10, 2004:
Added a comparison between generated byte code and generated source code with the tail recursion optimization.

September 28, 2004:
Determine the effect of a variety of changes to compilation and code generation.

November 18, 2004:

The results of making tail calls directly on the java stack, when possible, were added to the results section.

May 17, 2005:

- Effects of volatile quit flag in lecc.
- Effects of fixing space usage problem caused by strictness optimizations.

May 19, 2005:
The changes in performance caused by checking the quit flag inside tight tail recursive loops were added to the results.

June 7, 2005:
Added changes to performance caused by reintroducing eager evaluation of strict arguments in tail calls in lecc.
Added changes to performance caused by doing optimistic reduction in the evaluation loop in lecc.

June 9, 2005:
Added changes to performance caused by directly creating data constructor instances in lazy contexts.

June 20, 2005:
Added changes to performance resulting from optimizations to reducing over-saturated application chains and reducing graph traversals.

July 6, 2005:
Added performance changes resulting from caching boxed values.

August 11, 2005:
Add the results of the expression lifting optimization in the lecc machine.

September 19, 2005:
Add the results of creating primitive operations for bitwise operations on Int and Long.

December 7, 2005:
Add the results of lifting let-variable definitions.

March 24, 2006:
Added the results from a series of changes to the lecc runtime.

April 6, 2006:
Add the results from the unboxed return value optimization.

