Improving SCHEME Program Performance through Heuristic Redundant Predicate Elimination using PAC Learning Techniques

Michael R. Blair

December 9, 1993

Abstract

Functional programming languages encourage a modular style of programming in which large programs are composed of several small, independent, robustly designed modules. For the dynamically typed subset of the class of functional languages, this generally entails redundant type checks at run time. Contemporary attempts to mitigate this problem generally resort to compile-time *abstract interpretation* of programs to detect and eliminate these redundancies. Unfortunately, such techniques quickly become computationally unwieldy for large modular software systems.

This paper explores the combination of run-time code profiling and machine learning techniques (*viz.,* PAC learning) as a possible avenue for alleviating this difficulty. Specifically, run-time profiling is used to identify those code fragments that might best benefit from the elimination of redundant predicates while identifying a sample space of example code usage witnessed in actual code runs. These samples are used to hypothesize specific predications that show evidence of potential redundancy. The semantics of the language can then be employed to validate or reject these hypotheses. In this way, the profiler is used as both a focussing mechanism and as a sample-space "oracle" through which a heuristic estimate of redundant predicates can be identified.

The principle result of this paper is that heuristically identifying potentially computationally significant redundant predicates in this way can be done efficiently. Moreover, this heuristic approach is effective to the extent that it noticeably improves performance of an important class of SCHEME programs. It also offers a novel application of PAC learning in a setting not traditionally addressed by the machine learning research community.

1 Introduction: Functional Program Specialization

Functional programming languages have been continually celebrated as an auspicious way to code large complex software systems that are elegant, modular, and robust [Abelson *et al.* 88] [Field & Harrison 88]. Particularly touted is the elegance with which very subtle numerical control structures can be expressed in such languages [Halfant & Sussman 87/88] [Roylance 87/88]. Unfortunately, this elegance comes at some expense in performance. In essence, this elegance is achieved through the liberal use of higher order procedures and generic arithmetic operators. Specifically, the availability of higher order procedures encourages decomposing large problems into small, bite-sized code chunks that can be composed to build complex patterns of interaction. Generic arithmetic further encourages a very general style of coding which emphasizes the mathematical underpinnings of what is being expressed, avoiding distracting details of machine representations of numbers.

Current state-of-the-art techniques for reducing this overhead predominantly entail compile-time specialization of these very general programming mechanisms [Berlin 89] [Berlin 90] [Berlin & Weise 89] [Weise & Ruf]. This compile-time specialization, or *partial evaluation*, is achieved through a non-standard compile-time interpretation of the program, a technique known as *abstract interpretation* [Nielson 86] [Nielson 87/88]. Unfortunately, compilers that incorporate these techniques are very slow, sufficiently so that modern commercial production compilers do not resort to partial evaluation. The performance gain in the resultant code simply does not justify the cost of compilation. The principle reason for this is that abstract interpretation is an expensive symbolic evaluation process. If left unchecked, it can waste tremendous resources pursuing optimizations that will not prove relevant in practice. This problem is generally recognized as *the* principal shortcoming of partial evaluation (witness: [Launchbury 91]).

2 Run-Time Profiling Facilitates Efficient Specialization

To address this problem, several projects have pursued the use of run-time gathered profiling information to focus the specialization on only performance-critical code fragments [Samples 91] [Hansen 74] [Hansen & Wulf 76]. Such run-time based code optimization issues are becoming increasingly popular, at least among speculative researchers [Keppel, Eggers & Henry 91] [Wall 91]. One of the more successful efforts along these lines ([Ellis 86]) focusses primarily on identifying those predominant traces through the code which account for most of the execution time. These traces are efficiently identifiable through a variety of tracing mechanisms, *e.g.,* [Ball & Larus 92] [Larus 90]. Their extension to functional programming languages has recently been addressed [Appel, Duba & MacQueen 88].

Alas, these efforts generally involve compile-time analysis of code blocks in order to determine profile probe placement, again exacerbating compiler overhead. The natural alternative is to resort to a more statistical profile of code usage, so-called *PC sampling*, which periodically probes the program counter in the executing machine to determine the identity of the currently executing code block. With a sufficiently frequent periodic sample rate and a sufficiently long running execution, a reasonably accurate localization of time-critical code blocks can be achieved at modest run-time cost.[1]

3 Our Goal: Redundant Predicate Elimination

Once the predominant traces through the code have been identified, focussed code specialization can proceed. Specifically, code blocks that are determined to be performance bottlenecks can be expanded to reveal opportunities for specialization. For instance, consider the following program to compute Fibonacci numbers in the SCHEME [IEEE 91] dialect of LISP:

```
(define (fib-x z)
  "Fibonacci -- exponential algorithm  [cheat: identity on z < 0; fib on z > 0]"
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
         (fib-x (-   z 2)))))
```

Figure 1: Simple tree-recursive Fibonacci

Were profiling to reveal this procedure to be a bottleneck within some larger application,[2] specializing the generic arithmetic procedures comprising its body would be justified. For example, the generic `<` predicate expands as shown in Figure 2 in MIT SCHEME [Hanson 91].

Notice the plethora of type checks sprinkled throughout the code. Now imagine the redundancy involved in expanding the code for the remaining arithmetic operators (`+`, `-1+`, and `-`)! Fully expanding `fib-x` in this manner yields the monstrosity listed in Appendix A. This sort of predicate redundancy has been identified recently [Henglein 92] as accounting for as much as 60% of typical SCHEME programs. It is the recognition of these redundant predicates which the remainder of this paper addresses. Our first step towards re-casting this problem as a machine learning recognition problem is to re-formulate the input (SCHEME expressions) in a more mathematically accessible format.

[1] For instance, a sample period of 20 milliseconds on an HP 715 running MIT SCHEME was empirically found to produce a quantization error of approximately 10% with a slow-down of also around 10% for a variety of application programs [Blair 93].

[2] Admittedly, Fibonacci is unlikely to arise in practical applications, but very similarly structured arithmetic icons like `gcd` *are* common. Thus, without sacrificing relevance, I will proceed with Fibonacci as our running code example, principally because it is a minimally sufficiently rich example to carry the discussion while being a sufficiently familiar algorithm to not distract the reader.

```
;;; Generic <

(define (complex:< x y)
  (if (recnum? x)
      (if (recnum? y)
          (real:< (rec:real-arg '< x) (rec:real-arg '< y))
          (real:< (rec:real-arg '< x) y))
      (if (recnum? y)
          (real:< x (rec:real-arg '< y))
          ((copy real:<) x y))))

(define (rec:real-arg name x)
  (if (not (real:zero? (rec:imag-part x)))
      (error:wrong-type-argument x false name))
  (rec:real-part x))

(define (real:< x y)
  (if (flonum? x)
      (if (flonum? y)
          (flo:< x y)
          (rat:< (flo:->rational x) y))
      (if (flonum? y)
          (rat:< x (flo:->rational y))
          ((copy rat:<) x y))))

(define (rat:< q r)
  (if (ratnum? q)
      (if (ratnum? r)
          (int:< (int:* (ratnum-numerator q) (ratnum-denominator r))
                 (int:* (ratnum-numerator r) (ratnum-denominator q)))
          (int:< (ratnum-numerator q) (int:* r (ratnum-denominator q))))
      (if (ratnum? r)
          (int:< (int:* q (ratnum-denominator r)) (ratnum-numerator r))
          (int:< q r))))

;; flonum:< and int:< are "microcoded" primitives in MIT CScheme as follows:

#define FLONUM_BINARY_PREDICATE(operator)                               \
{                                                                       \
  PRIMITIVE_HEADER (2);                                                 \
  Set_Time_Zone (Zone_Math);                                            \
  BOOLEAN_RESULT ((arg_flonum (1)) operator (arg_flonum (2)));          \
}
DEFINE_PRIMITIVE ("FLONUM-LESS?", Prim_flonum_less_p, 2, 2, 0)
     FLONUM_BINARY_PREDICATE (<)

Boolean
DEFUN (integer_less_p, (n, m), SCHEME_OBJECT n AND SCHEME_OBJECT m)
{
  return
    ((FIXNUM_P (n))
     ? ((FIXNUM_P (m))
        ? (FIXNUM_LESS_P (n, m))
        : (BIGNUM_LESS_P ((FIXNUM_TO_BIGNUM (n)), m)))
     : (BIGNUM_LESS_P (n, ((FIXNUM_P (m)) ? (FIXNUM_TO_BIGNUM (m)) : m))));
}
```


Figure 2: Full Expansion of MIT Scheme Generic <

4 Semantics of Scheme Conditionals: From Expressions into DAGs

The formal denotational semantics [Stoy 77] of the SCHEME `if` expressions is [Clinger & Rees 91] [IEEE 91]:

$$\mathcal{E}\,[\![(\texttt{if}\ E_p\ E_c\ E_a)]\!]\ \rho\ \sigma\ \kappa\ =\ \mathcal{E}\,[\![E_p]\!]\ \rho\ \sigma\ (\lambda\ (v\ \sigma')\ \ (\text{if}\ v\ \ \begin{array}{ll}\text{then} & \mathcal{E}\,[\![E_c]\!]\ \rho\ \sigma'\ \kappa\\ \text{else} & \mathcal{E}\,[\![E_a]\!]\ \rho\ \sigma'\ \kappa))\end{array}$$

Here, $\mathcal{E}$ takes a (quasi-quoted/bracketed) SCHEME expression and evaluates it in the given environment (ρ), store (σ), and continuation (κ). Essentially, the predicate expression is evaluated in the given environment and store. This produces a value (v) and a (possibly modified) store (σ'), which are passed to an explicit continuation λ-expression. This λ-continuation then proceeds, based on the Boolean value of the predicate, either to evaluate the consequent or the alternative expression in the given environment with the store resulting from the evaluation of the predicate expression. The result of the consequent or alternative evaluation is then passed to the initially specified continuation (κ).

It is not difficult to observe that this continuation-based denotational semantics, in effect, flattens nested conditional expressions into directed acyclic code flow graphs. To wit, one can envision the denotational semantics as performing an explicit conversion of the input SCHEME syntax into a so-called Continuation Passing Style (CPS) [Steele 78]. For instance, the following nested predication:

```
(if (if pred i-con i-alt) o-con o-alt)
```

CPS converts into the following (ignoring ρ and σ):

```
(CPS-if pred i-con i-alt (lambda (v) (CPS-if v o-con o-alt k_return)))
```

where

$$\begin{array}{rcl}\mathcal{E}\,[\![(\texttt{CPS-if}\ E_p\ E_c\ E_a\ E_\kappa)]\!] & = & ((\lambda\ (v\ \kappa)\ \ (\text{if}\ v\ \ \begin{array}{ll}\text{then} & \mathcal{E}\,[\![E_c]\!]\ \kappa\\ \text{else} & \mathcal{E}\,[\![E_a]\!]\ \kappa))\end{array}\\ & & \ \ \mathcal{E}\,[\![E_p]\!]\\ & & \ \ \mathcal{E}\,[\![E_\kappa]\!])\end{array}$$

This standard semantic transformation enables us to view *any* conditional expression in SCHEME as expressible by an equivalent directed acyclic graph (DAG) with only primitive predicates at the nodes (noting that all other conditional forms in SCHEME can be de-sugared into `if` expressions, including `and`, `or`, and `not` [Clinger & Rees 91]). In this way, we can view any SCHEME program as essentially denoting a control flow DAG, each node of which identifies a conditional predicate with the branches emanating from each branch identifying the consequent and alternative continuations.

5 Recognition of Redundant Predicates

Taking the DAG re-formulation of Scheme programs outlined above, we can cast redundant predicate detection as a PAC learning problem [Valiant 84] over Boolean DAGs fairly readily. First, associate with each predicate node in the control flow DAG a distinct Boolean variable. For a Scheme program of textual length n, there can be at most n such nodes and, hence, there will be at most n Boolean variables. Given that predicates will be decomposed (expanded) into their constituent primitives, the identification of redundant predicates can be seen as identifying nodes in the control flow DAG whose truth values correlate (positively or negatively) with nodes lower in the DAG.

For instance, the following Boolean DAG corresponds roughly to the Scheme expression `(if (if p1 c2 a3) c4 a5)`, where `p1`, `c2`, and `a3` are predicates (internal nodes) and `c4` and `c5` are result expressions (leaves):

```
             .
            / \
           ( 1 )
         T /\ /\ F
         . /  '  \ .
        / \       / \
       ( 2 )     ( 3 )
     T /\ /\ F T /\ /\ F
      /  '  \   /  '  \
      \        /       /
       \      /       /
        \    /  \    /
         \  /    \  /
         [4]     [5]
```

Figure 3: Boolean Decision DAG for `(if (if p1 c2 a3) c4 a5)`

In this setting, redundant predicates are recognized as correlated branches in the Boolean DAG. For instance, if every time <1> branches true we observe that <2> likewise branches true whereas when <1> branches false then <3> branches true, this would be evidence that predicate <1> is correlated with predicate <2> and reverse-correlated with <3>. On the other hand, if when <1> branches false <3> is sometimes observed to branch true and sometimes false, then no correlation (positive or negative) could exist between predicates <1> and <3>. In general, any node can correlate with (dominate) any node below it in the DAG.

One could always simply grovel over the entire DAG using an abstract interpreter to determine these correlations, but this would quickly become expensive for complex, multi-node DAGs. In particular, the number of possible correlations can grow as n^2 since, at worst case, a decision list [Rivest 87] (simple chain vs. tree) of nodes can admit each node potentially correlating with each node to its right, as depicted in Figure 4. Considering that abstract interpretation is usually expensive since it requires a generally symbolic semantic entailment evaluation of Scheme primitives [Nielson 86], it is desirable to avoid as many

```
  .      .      .      .               .
 / \  T / \  F / \  T / \  T       F  / \  F
( 1 )---( 2 )---( 3 )---( 4 )--- ... ---( n )---[z]
 \ /    \ /    \ /    \ /             \ /
  |      |      |      |               |
  | F    | T    | F    | F             | T
 [a]    [b]    [c]    [d]             [y]
```

Figure 4: Boolean Decision DAG Chain (a.k.a. Decision List)

such entailment tests as possible.

This is where an efficient PAC learning approach can come in handy. Specifically, recall that in order to identify the potential bottlenecks in the code, we will have already randomly sampled the program counter some number of executions of various code blocks. Those blocks which were identified as being frequently sampled will naturally have been observed executing a substantial number of times. To wit, each time a block is PC sampled, there is an opportunity to peek at its activation record on the call stack and squirrel away a copy of the argument(s) on which it was called. Later, if a block is deemed a bottleneck, we can draw on these argument samples to generate more closely monitored example executions of the block.

Given a sufficient number of examples of code executions through the control DAG, we should hope to develop an accurate hypothesis of precisely where the predicate correlations exist. These hypotheses, of course, can err in two ways: first, we can hypothesize a correlation which in fact does not semantically exist. This can occur if every observed example execution happens to follow some correlated path yet the semantics of the predicates so hypothesized to be correlated do not logically entail. Therefore, whatever hypothesized correlations are produced must be checked against the semantics of the predicates involved before using the suspected correlations to eliminate code from the SCHEME source— in short, we can use our hypothesized correlations as a focussing mechanism for determining logical entailment under a suitable abstract interpretation of the predicates but we cannot blithely accept them as gospel and wrecklessly eliminate code without first verifying the hypothesis.

The second manner in which we can err is far less troublesome: we can fail to detect a predicate correlation which in fact *does* exist in the control DAG. Given that we observe actual code executions in formulating our hypotheses, this can occur only when there is some correlation along a path which we, in fact, never witness being traversed. This can arise either because some intervening correlation forces branching away from the path of interest— in which case we don't care since the undetected correlation is within "dead code" anyway— or because the un-exercised path happens not to have been navigated by the witnessed example executions. In the latter case, we may fail to detect a redundancy that could in theory lead to additional code improvement, yet the fact that we miss this opportunity is unlikely to have much bearing on the code execution in practice *if it can be argued that the undetected redundancy is ever likely to be executed anyway.*

At this point, it should be abundantly clear that what we would like is an efficient PAC algorithm to develop sufficiently accurate hypotheses of predicate correlations using a reasonable number of samples such that PC sampling is likely to be able to provide a realistic distribution of actual code executions. The next section re-formulates this as a PAC problem.

6 Decision DAG Node Correlation as a PAC Problem

Given a decision DAG comprised of n nodes, we wish to hypothesize correlations of node values. Thus, our hypothesis class, H, is the set of all possible Boolean variable correlations for any decision DAG of n Boolean variables, one variable per node. Notice that in any particular learning instance where we try to hypothesize a set of node correlations, we will be given the decision DAG under consideration, meaning that we know the full DAG structure of nodes and branches. What we are attempting to learn are the correlations of these nodes in the fully revealed DAG.

6.1 Sample Bound on PAC Learning Decision DAGs

For any particular fixed DAG of n nodes, any particular node n_i can correlate with any other node occurring below it in the DAG structure. For each such potential correlated descendant node, we must try to ascertain if a correlation exists. The set of all possible correlations for each node of an n-node decision DAG calculated as follows:

For the root node, it could correlate with any of the other $n-1$ nodes in the decision DAG positively, negatively, or not at all. For each internal node, that node can correlate with any descendant node. The worst case number of potential correlated nodes (descendants) arises when the decision DAG is a decision list, that is, a simple chain of nodes (*e.g.,* Figure 4). In this case, the total number of potential correlations is:

$$3^{n-1} \cdot 3^{n-2} \cdot \dots \cdot 3^{1} \;=\; 3^{O(n^2)}$$

This bounds the cardinality of the hypothesis class as $\mid \mathcal{H} \mid \leq \; 3^{O(n^2)}$. Using a standard Occam Algorithm [Blumer *et al.* 87], this leads us to a bound on the number of samples needed to PAC learn an instance of this hypothesis class of $m \;\geq\; O(\frac{1}{\epsilon}(n^2 + \ln(\frac{1}{\delta})))$. Thus, the class of decision DAGs of n nodes (variables) is efficiently PAC learnable.

6.2 The Minimally Consistent Decision DAG Algorithm

The detail remains, however, of how we find a minimal consistent hypothesis for a given set of samples. It also remains, for that matter, just what the characterization of a sample is within this representation.

First, how to represent samples. Our oracle for generating examples is constructed out of actual PC-sampled activations of the SCHEME code block whose decision DAG node correlations we are trying to learn, as suggested in the previous section. Specifically, when we desire a fresh example, we just wait around for the PC sampler to sample another call to the target SCHEME code block.[3] By laying probes on the exit points from the code block, we can detect the corresponding leaf in the decision dag which is ultimately reached by the given sample. Each leaf uniquely determines the path from root to leaf that was traversed during the execution. As such, each path reflects a witness of true/false assignments to each node touched during the path traversal. This corresponds to a conjunction of literals taken from the set of literals over the n Boolean variables associated with the n nodes in the decision DAG. In sum, therefore, each sample is expressed as a conjunction of literals taken from the $2n$ literals naturally associated with the n variables (nodes) in the decision DAG.

Now, how then do we collapse these conjunctions into a hypothesis of node correlations? We associate with each variable (node) two sets: $\mathcal{P}$ and $\mathcal{N}$, both initially empty. With each new example conjunction of literals, we proceed as follows. Assume, without loss of generality, that the literals in the conjunction appear in order from left-to-right with respect to the order of the nodes traversed in the sample execution path within the decision DAG. Now, start at the root and walk the literals in the conjunction from left to right. For each literal, check if the variable associated with that literal is in the $\mathcal{N}$ set for the root node. If so, that variable is known to be non-correlated with the root node, so simply continue walking the literals. Otherwise, check if the inverse of the literal appears in the set of potential correlates $\mathcal{P}$ for the root node. If so, remove the inverse literal from $\mathcal{P}$ and add the associated variable to $\mathcal{N}$: we have seen both a positive and a negative association between this variable and the root node so they cannot be correlated. Once the full conjunction has been processed in this manner for the root node, we can proceed to the node associated with the variable of the first literal in the conjunction. We then process the remainder of the conjunction (*i.e.,* the conjunction minus the first literal) with respect to this new node, checking against this new node's $\mathcal{N}$ and potentially adding to or removing from it's $\mathcal{P}$ as we did for the root node. This algorithm takes time at most $O(n^2)$ for each sample conjunction since each sample can have at most n literals, which requires n traversals of its successive tails, one traversal for each depicted node in the literals present.

When the algorithm has completed processing the m examples, any non-correlations will be depicted by elements in the $\mathcal{N}$ sets of each node. Each positive correlation hypothesized will be expressed as a positive literal within the correlated node's $\mathcal{P}$ set; each negative correlation, as a negated literal. Each node not actually traversed by any of the samples will not appear in any node's $\mathcal{N}$ nor in any node's $\mathcal{P}$, either negated or not. Therefore, the hypothesized correlations correspond to the union over all nodes' $\mathcal{P}$ of the hypothesized correlation of that node with each element of its final $\mathcal{P}$. Given the polynomial bound on the computation needed by each sample, this algorithm is efficient. Thus, node correlation of decision DAGs is efficiently PAC learnable using this standard Occam algorithm.

[3] I'll return to this somewhat odd sort of oracle in the next section. Bear with me.

7 On-Line Incremental PAC Learning with Target Concept Drift

For a given distribution D of sample executions, therefore, we should be able to find an arbitrarily accurate hypothesis of the potential predicate correlations in the corresponding SCHEME expression. For our PC-sample derived distribution this could be construed as a somewhat odd notion of an example oracle since we must wait around passively for the program user to issue a sufficient number of procedure calls to witness the needed number of samples. At first thought this may seem unnerving, but let me try to address the possible concerns.

7.1 On-Line Incremental PAC Learning

First, our setting is one of an on-line learning task where we are attempting to learn properties of a program by witnessing instances of its use during actual program runs. For instance, I ultimately envision a system where a user loads in some compiled SCHEME code and plays with it as usual. As this interaction proceeds, a period PC-sample interrupt is issued, unobservable by the user, which keeps an eye on what code is being executed. When a sufficient number of samples have been taken that a) the quantization error of the sampler has been mitigated,[4] and b) some set of code blocks have been found to exhibit an interesting threshold of utilization,[5] each apparently interesting code block can then be code expanded to the depth and extent appropriate to its margin above threshold. For instance, if `fib-x` accounted for an impressive percentage of samples (after a sufficient number of samples have been taken that we trust these utilization stats) then we may see fit to fully expand its source code all the way down to the primitives, as demonstrated in Appendix A.

Given these expanded target blocks, we can compute from their expanded source code the number of predicate nodes in their associated decision DAGs. From this, we compute a number of samples desired of each block before we consider ourselves confident in our hypothesized node correlations. If the requisite number of samples is encountered for a particular code block of interest, we can then proceed to generate a hypothesized node correlation, verify it against the actual language semantics, then tune the code appropriately to eliminate redundant predicates and prune unreachable code fragments. Once the verification has been performed each node's $\mathcal{P}$ and $\mathcal{N}$ sets can be adjusted to reflect the semantics of the actual predicates.

As the code continues to run we can continue to PC sample its execution. Over time, varying code blocks may appear interesting and subsequently may be expanded and tuned. Were our tuning of code not conservative in the sense that every code change is verified safe against the language semantics, this could be cause for alarm. Despite its semantic safety,

[4]This is a function of the sample frequency relative to the expected execution duration of a typical sized code block. This is an architecture and language implementation specific empirically calculated threshold.

[5]In sample programs profiled with the MIT SCHEME PC sampler I implemented last summer, a reasonable threshold seems to be any code block that accounts for more than, say, 10% of the samples taken [Blair 93].

there is nevertheless a very practical concern: if we allow every code block of transient interest to be expanded in size for purposes of tuning then we would expect to eventually run out of running memory in which to maintain this ever growing code. To mitigate this difficulty, a practical implementation of this scenario would need to enforce some space reclamation policy, un-expanding code which has not ultimately demonstrated its utility over a substantial time frame. Devising a workable reclamation policy will certainly become paramount to continuing this work.

7.2 Distribution Drift and Error/Accuracy Containment

This issue of code blocks transiently seeming to be of interest is actually symptomatic of a more general phenomenon. In our conservative learning scenario, where we learn ultimate truths of the correlations of fixed predicates within a static semantics, the worst we can do is waste resources. That is, if our training examples exhibit a particular distribution but that distribution changes over time after we have "finished" training, then the accuracy of our hypothesis about the target concept could degrade. Above, I suggested using a more on-line continual re-training scenario to try to accommodate this sort of distribution drift. In general, the accuracy of a hypothesis being used to predict instances of a target concept could be monitored. Whenever the error rate ϵ of the hypothesis is observed to be distressingly higher than the accuracy prediction δ, a selection of recent faulty predictions could be fed back into the learning algorithm to try to further tune the hypothesis.

7.3 Target Concept Drift

The problem can be much more severe if it is not the distribution that drifts but the target concept itself. For example, take the obvious algorithm for PAC learning simple conjunctions over n variables [Kearns & Vazirani 94, Sec. 2.3] Initially, we hypothesize the conjunction of all literals, both positive and negative, hence starting with the hypothesis "false". For each conjunction of literals presented as being a satisfying truth assignment for the target conjunction (*i.e.,* given a sequence of positive examples), we update our hypothesis by cancelling literals which contradict the positive examples given. Once a literal is cancelled by this method, it is gone forever, never to be re-introduced into the hypothesis conjunction. If, over time, an adversary decides to shift the actual target, such an irreversible (monotonic) algorithm could be driven to fail to produce any hypothesis at all.

Such target drifts might at first glance appear to be expressible as a specific class of noise in the oracle, but the problem is in determining a sufficiently large sample to try to deal with this noise. Consider, for the moment, how such noise is dealt with in a statistical query model [Kearns 93]. Informally, the closest thing to handling concept drift in this model is to compel the oracle to reveal information, say, about the nature of the target drift. This is suggestive of presenting a sufficient number of examples of drift to admit the learner to, in some sense, learn what the drift is like. This analogy may work for some particularly well-behaved classes of drift, for for restless moving target there seems to be no hope.

Imagine, for example, if a particularly adversarial concept drift shifted with each example such that every new example you are given is immediately followed by a completely contradictory example. No learning would be possible in such a horrid situation. If, however, we constrained the adversary to switch the target only at a fixed maximum frequency, some learning could be possible in an on-line incremental setting. For instance, we could retain good error rate and accuracy in face of this moving adversary if we limited the adversary to shift the target only after we have performed sufficiently well in the recent past to redeem our error rate and accuracy. That is, we start with an initial sample which is computed based on the characteristics of the hypothesis class, such as its cardinality or VC-dimension [Blumer *et al.* 89].[6] We would then proceed with the deduced hypothesis. Once a sufficient number of successful predictions has been demonstrated using this hypothesis to establish the projected accuracy of the hypothesis, we could then allow the adversary to shift the target concept. The adversary would then be compelled to hold that target until we adjust sufficiently. For instance, once our accuracy has degraded beyond, say, one standard deviation of what was predicted, we could abandon our hypothesis, collect examples to formulate a new hypothesis, then stick with that new hypothesis until we have again established our projected accuracy over the entire history of predictions. At the point, the adversary could again be permitted to shift the target. Under such a constraint, we in essence compel the adversary to cooperate in allowing us to "save face" after the target shift has degraded our performance.

Whether or not this is a realistic constraint to impose is unclear. In the setting of code profiling, however, there *is* the possibility of compelling the adversary to cooperate in a different sense. Consider, for example, a program which is called from several distinct contexts. The characteristic utilization of the program may differ substantially in the different contexts, so much so that little performance tuning could be done on the code block in isolation. If, on the other hand, we compel the oracle (PC sampler) to identify the caller of this code block whenever we are presented with a sample of its use, then we could be careful to generate separate independent utilization hypotheses for the distinct callers. This is not an unrealistic demand since, at the time of the sample, the activation record for the sampled activation of the code block will, of necessity, include a return address pointer to the caller.[7]

[6] Note that we do *not* let the hypothesis class drift: we allow the target concept to drift *within the confines of the hypothesis class*. To allow the hypothesis class to drift would be tantamount to allowing the target to drift within a hypothesis class that spans all hypothesis classes which one might want to drift into. To broaden this hypothesis class too liberally would preclude PAC learnability altogether.

[7] Well, modulo tail recursion, anyway. Even then, though, a distinguishable return address to the continuation will be provided, even if this continuation isn't the caller proper.

8 Conclusions

To summarize, I have laid out, in some detail, how SCHEME programs could be performance tuned by using PC sampling information and execution probes to focus the search for redundant predicates. It has been argued that this is a productive goal. Indeed, hand-tuned code along these lines does show noticeable performance improvement. Specifically, returning to the tree recursive Fibonacci example earlier, hand-specializing all generic arithmetic operators to simple integer operators improved the code performance by 15%. Although not a staggering improvement, it forebodes well.

Moreover, it has been argued, at least, that this sort of performance tuning is achievable efficiently through the cooperation of an inexpensive statistical program profiler (PC sampler) and an efficient Occam-style learning algorithm for hypothesizing key points of potentially beneficent code tuning.

In retrospect, it has been interesting, for me at least, to contemplate how PAC learning techniques can be applied towards generating efficient and tunable heuristics for focussing code optimization. The basis of the approach has been to recognize that a complete search for program optimization over the entire space of program code is, in general, too expensive to justify the resultant gain in performance. Therefore, rather than attempt to generate provably optimal code (in some sense), it can be productive to instead generate an inexpensive yet well-focussed heuristic approximation to where optimization techniques are likely to be beneficial. Generating such a hypothesis of where to focus efforts can then be readily verified by substantially reduced overhead than the more exhaustive brute-force compile-time techniques typical of static code optimization strategies. Whether this is truly a practical approach to SCHEME code tuning has yet to be demonstrated since a complete implementation of this strategy is still under development. Initial measurements, nevertheless, look somewhat encouraging.

A Fully Expanded Fibonacci

```
(define (fib-x z)
  "Fibonacci -- exponential algorithm  [cheat: identity on z < 0; fib on z > 0]"
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
 (fib-x (-   z 2)))))
```

$\Longrightarrow$

```
(define (moby-fib-x z)
  "Fibonacci -- exponential algorithm  [cheat: identity on z < 0; fib on z > 0]"
  (if (if (recnum? z)                                       ; ** IN-LINED (COMPLEX:< z 2)
          (if (recnum? 2)
              (real:< (rec:real-arg '< z) (rec:real-arg '< 2))
              (real:< (rec:real-arg '< z) 2))
          (if (recnum? 2)
              (real:< z (rec:real-arg '< 2))
              (if (flonum? z)                               ; ** IN-LINED (REAL:< z 2)
                  (if (flonum? 2)
                      (flo:<                 z  2)
                      (rat:< (flo:->rational z) 2))
                  (if (flonum? 2)
                      (rat:< z (flo:->rational 2))
                      (if (ratnum? z)                       ; ** IN-LINED (RAT:< z 2)
                          (if (ratnum? 2)
                              (int:< (int:* (ratnum-numerator   z)
                                            (ratnum-denominator 2))
                                     (int:* (ratnum-numerator   2)
                                            (ratnum-denominator z)))
                              (int:< (ratnum-numerator z)
                                     (int:* 2 (ratnum-denominator z))))
                          (if (ratnum? 2)
                              (int:< (int:* z (ratnum-denominator 2))
                                     (ratnum-numerator 2))
                              (int:< z 2)))))))
      z
      (let ((fib_n-1 (moby-fib-x
                      (if (recnum? z)                       ; ** IN-LINED (COMPLEX:-1+ z)
                          (make-recnum (real:-1+ (rec:real-part z)) (rec:imag-part z))
                          (if (flonum? z)                   ; ** IN-LINED (REAL:-1+ z)
                              (flo:- z flo:1)
                              (if (ratnum? z)               ; ** IN-LINED (RAT:-1+ z)
                                  (let ((z* (ratnum-denominator z)))
                                    (make-ratnum (int:- (ratnum-numerator z) z*) z*))
                                  (int:-1+ z))))))
            ⋮
            [computation of fib_n-2 and sum of fib_n-1 with fib_n-2 follow]
            ⋮
```

```
   (fib_n-2 (moby-fib-x
             (if (recnum? z)                          ; ** IN-LINED (COMPLEX:- z 2)
                 (if (recnum? 2)
                     (complex:%make-rectangular
                      (real:- (rec:real-part z) (rec:real-part 2))
                      (real:- (rec:imag-part z) (rec:imag-part 2)))
                     (make-recnum (real:- (rec:real-part z) 2)
                                  (rec:imag-part z)))
                 (if (recnum? 2)
                     (make-recnum (real:-    z (rec:real-part 2))
                                  (real:negate (rec:imag-part 2)))
                     (if (flonum? z)                  ; ** IN-LINED (REAL:- z 2)
                         (if (flonum? 2)
                             (flo:- z              2 )
                             (flo:- z (rat:->flonum 2)))
                         (if (flonum? 2)
                             (flo:- (rat:->flonum z) 2)
                             (if (ratnum? z)          ; ** IN-LINED (RAT:- z 2)
                                 (if (ratnum? 2)
                                     (let ((u  (ratnum-numerator   z))
                                           (u* (ratnum-denominator z))
                                           (v  (ratnum-numerator   2))
                                           (v* (ratnum-denominator 2)))
                                       (let ((d1 (int:gcd u* v*)))
                                         (if (int:= d1 1)
                                             (make-rational (int:- (int:* u v*)
                                                                   (int:* v u*))
                                                            (int:* u* v*))
                                             (let* ((u*/d1 (int:quotient u* d1))
                                                    (t
                                                     (int:- (int:* u (int:quotient v* d1))
                                                            (int:* v u*/d1))))
                                               (if (int:zero? t)
                                                   0 ;(make-rational 0 1)
                                                   (let ((d2 (int:gcd t d1)))
                                                     (make-rational
                                                      (int:quotient t d2)
                                                      (int:* u*/d1
                                                             (int:quotient v* d2)))))))))
                                     (let ((u  (ratnum-numerator   z))
                                           (u* (ratnum-denominator z)))
                                       (make-rational (int:- u (int:* 2 u*)) u*)))
                                 (if (ratnum? 2)
                                     (let ((v  (ratnum-numerator   2))
                                           (v* (ratnum-denominator 2)))
                                       (make-rational (int:- (int:* z v*) v) v*))
                                     (int:- z 2))))))))))
⋮
[sum of fib_n-1 with fib_n-2 follows]
⋮
```

```
(if (recnum? fib_n-1)                             ; ** IN-LINED (COMPLEX:+ FIB_N-1 FIB_N-2)
    (if (recnum? fib_n-2)
        (complex:%make-rectangular
         (real:+ (rec:real-part fib_n-1) (rec:real-part fib_n-2))
         (real:+ (rec:imag-part fib_n-1) (rec:imag-part fib_n-2)))
        (make-recnum (real:+ (rec:real-part fib_n-1) fib_n-2)
                     (rec:imag-part fib_n-1)))
    (if (recnum? fib_n-2)
        (make-recnum (real:+ fib_n-1 (rec:real-part fib_n-2))
                     (rec:imag-part fib_n-2))
        (if (flonum? fib_n-1)                     ; ** IN-LINED (REAL:+ FIB_N-1 FIB_N-2)
            (if (flonum? fib_n-2)
                (flo:+ fib_n-1               fib_n-2 )
                (flo:+ fib_n-1 (rat:->flonum fib_n-2)))
            (if (flonum? fib_n-2)
                (flo:+ (rat:->flonum fib_n-1) fib_n-2)
                (if (ratnum? fib_n-1)             ; ** IN-LINED (RAT:+ FIB_N-1 FIB_N-2)
                    (if (ratnum? fib_n-2)
                        (let ((u  (ratnum-numerator   fib_n-1))
                              (u* (ratnum-denominator fib_n-1))
                              (v  (ratnum-numerator   fib_n-2))
                              (v* (ratnum-denominator fib_n-2)))
                          (let ((d1 (int:gcd u* v*)))
                            (if (int:= d1 1)
                                (make-rational (int:+ (int:* u v*) (int:* v u*))
                                               (int:*       u*           v*   ))
                                (let* ((u*/d1 (int:quotient u* d1))
                                       (t
                                        (int:+ (int:* u (int:quotient v* d1))
                                               (int:* v               u*/d1 ))))
                                  (if (int:zero? t)
                                      0 ;(make-rational 0 1)
                                      (let ((d2 (int:gcd t d1)))
                                        (make-rational
                                         (int:quotient t d2)
                                         (int:* u*/d1 (int:quotient v* d2)))))))))
                        (let ((u  (ratnum-numerator   fib_n-1))
                              (u* (ratnum-denominator fib_n-1)))
                          (make-rational (int:+ u (int:* fib_n-2 u*)) u*)))
                    (if (ratnum? fib_n-2)
                        (let ((v  (ratnum-numerator   fib_n-2))
                              (v* (ratnum-denominator fib_n-2)))
                          (make-rational (int:+ (int:* fib_n-1 v*) v) v*))
                        (int:+ fib_n-1 fib_n-2))))))))))
```

References

[Abelson *et al.* 88]
Harold Abelson, Matthew Halfant, Jacob Katzenelson and Gerald Jay Sussman
The LISP *Experience*
Annual Reviews in Computer Science, Vol.3, 1988, pp.167–195

[Appel, Duba & MacQueen 88]
Andrew W. Appel, Bruce F. Duba, and David B. MacQueen
Profiling in the Presence of Optimization and Garbage Collection
CS-TR-197-88, Princeton University Department of Computer Science, 1988

[Ball & Larus 92]
Thomas Ball and James R. Larus
Optimally Profiling and Tracing Programs
In Proc. ACM POPL 92, Aug 1992, pp.59–70

[Berlin 89]
Andrew A. Berlin
A Compilation Strategy for Numerical Programs Based on Partial Evaluation
MIT S.M. Thesis, MIT AI-TR-1144, Jul 1989

[Berlin 90]
Andrew A. Berlin
Partial Evaluation Applied to Numerical Computation
In Proc. ACM LISP and Functional Programming, 1990.

[Berlin & Weise 89]
Andrew Berlin and Daniel Weise
Compiling Scientific Code using Partial Evaluation
MIT AI-Memo-1145, Jul 1989

[Blair 93]
Michael R. Blair
Author's Ph.D. research progress report (work in progress) Summer 1993.

[Blumer *et al.* 87]
Anselm Blumer, Andrzej Ehrenfreucht, David Haussler, and Manfred K. Warmuth
Occam's Razor
Information Processing Letters, Vol.24 No.6, April 1987, pp.377–380

[Blumer *et al.* 89]
Anselm Blumer, Andrzej Ehrenfreucht, David Haussler, and Manfred K. Warmuth
Learnability and the Vapnik-Chervonenkis Dimension
JACM Vol.36 No.4, October 1989, pp.929–965

[Clinger & Rees 91]
William Clinger and Jonathan Rees (Editors)
Revised^4 Report on the Algorithmic Language SCHEME
MIT/AI/Memo 848b, November 1991 or To appear in ACM LISP Pointers.

[Ellis 86]
John R. Ellis
Bulldog: A Compiler for VLIW Architectures
ACM Doctoral Dissertation Award 1985, The MIT Press, 1986

[Field & Harrison 88]
Anthony J.Field and Peter G.Harrison
Functional Programming
(International Computer Science Series)
Addison-Wesley, 1988

[Halfant & Sussman 87/88]
Matthew Halfant and Gerald Jay Sussman
Abstraction in Numerical Methods
MIT AI Memo 997, Oct 1987, or
In Proc. ACM Conference on LISP and Functional Programming, Jul 1988, pp.1–7

[Hansen 74]
Gilbert Joseph Hansen
Adaptive Systems for the Dynamic Run-Time Optimization of Programs
Ph.D. Dissertation, Carnegie-Mellon University, March 1974

[Hansen & Wulf 76]
G. Hansen and W. A. Wulf
Adaptive Code Optimization
In Program Optimization, Infotech State of the Art Report, 1976

[Hanson 91]
Chris Hanson
MIT SCHEME *Reference Manual*
MIT AI TR-1281, Nov 1991

[Henglein 92]
Fritz Henglein
Global Tagging Optimization by Type Inference
In Proc. ACM LISP and Functional Programming, 1992, pp.205–215.

[IEEE 91]
IEEE SCHEME Standardization Committee
IEEE Standard for the SCHEME *Programming Language*
IEEE Std 1178-1990, May 1991

[Kearns 93]
Michael Kearns
Efficient Noise-Tolerant Learning from Statistical Queries
In Proc. ACM STOC 93, pp.392–401.

[Kearns & Vazirani 94]
Michael J.Kearns and Umesh V.Vazirani
Topics in Computational Learning Theory
Unpublished preliminary draft manuscript (to be published early 1994)

[Keppel, Eggers & Henry 91]
David Keppel, Susan J. Eggers, and Robert R. Henry
A Case for Runtime Code Generation
TR 91-11-04, University of Washington, Department of Computer Science and Engineering

[Larus 90]
James R. Larus
Abstract Execution: A Technique for Efficiently Tracing Programs
In Software—Practice and Experience Vol. 20, No. 12, Dec 1990

[Launchbury 91]
John Launchbury
Projection Factorizations in Partial Evaluation
Distinguished Dissertations in Computer Science, Cambridge University Press, 1991

[Nielson 86]
Flemming Nielson
Abstract Interpretation of Denotational Definitions (A survey)
In STACS '86, Lecture Notes in Computer Science 210, Springer-Verlag, 1986, pp.1–20

[Nielson 87/88]
Flemming Nielson
Strictness Analysis and Denotational Abstract Interpretation (Extended Abstract)
In Proc. ACM POPL 1987, Jan 1987, pp.120–131, as condensed from
Information and Computation, Vol.76 No.1, Jan 1988, pp.29–92.

[Rivest 87]
Ronald L.Rivest
Learning Decision Lists
Machine Learning Vol.2 No.3, November 1987, pp.229–246

[Roylance 87/88]
Gerald L.Roylance
Expressing Mathematical Subroutines Constructively
MIT AI Memo No.999, Nov 1987, or
In Proc. 1988 ACM Conference on Lisp and Functional Programming, Jul 1988, pp.8–13

[Samples 91]
Alan Dain Samples
Profile-Driven Compilation
Ph.D. Dissertation UCB/CSD 91/627, UC Berkeley, Computer Science Division (EECS), Apr 1991

[Stoy 77]
Joseph E.Stoy
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory
(The MIT Press Series in Computer Science)
The MIT Press, 1977.

[Steele 78]
Guy Lewis Steele Jr.
RABBIT*: A Compiler for* SCHEME *(A Study in Compiler Optimization)*
MIT S.M. Thesis, MIT AI-TR-474, May 1978

[Valiant 84]
L.G.Valiant
A Theory of the Learnable
CACM Vol.27 No.11, Nov. 1984, pp.1134–1142

[Wall 91]
David W. Wall
Predicting Program Behavior Using Real or Estimated Profiles
In Proc. ACM PLDI 91, Jun 1991, pp.59–70

[Weise & Ruf]
Daniel Weise & Erik Ruf
Computing Types During Partial Evaluation
Stanford Fuse Memo 90-3-revised, Dec 1990