Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Cambridge, Massachusetts

Proposal for Thesis Research in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

Title: DESCARTES: A Dynamically Adaptive Compiler and Run-Time System using Continual Profil-Driven Program Multi-Specialization

Submitted by: Michael R. Blair
545 Technology Square #439
Cambridge, Ma.

Signature of Author:

Date of Submission: December 11, 1992
Expected Date of Completion: September 1, 1994
Laboratory where Thesis will be done: M.I.T. Laboratory for Computer Science

Brief Statement of Problem:

Continually profiling and adapting multiple specializations of a program as they are run on actual inputs can provide greater performance improvement at lower cost than traditional mainstream compile-time code optimization techniques.

I propose to demonstrate this claim by integrating a traditional optimizing compiler with a code profiler then augmenting the run-time system to use these profiling hooks for adaptive code optimization, using a cost-benefit investment strategy to govern the optimization efforts. Thus, focus for the optimizations is provided by profiling while throttling is provided by the cost-benefit investment model.

Why do I expect this to out-perform traditional compile-time optimization techniques? Traditional techniques for improving program performance rely on information deducible at compile time. Such information is static. It is used to improve the performance of a procedure on its entire domain of possible inputs. Recently developed compiler techniques have yielded further improvements by specializing multiple instances of a program for specific distinct inputs based on textually apparent invocations of the program from within other programs. This too uses only static information. The current state of the art in performance enhancement suggests that gathering information from sample runs of a program (a.k.a. "profiling") can provide still more valuable information. Such strategies gather dynamic information about a program's performance on a suite of typical inputs, re-compiling the program in light of this previously unavailable dynamic information to further improve the expected average performance of the program by improving its performance on the chosen input suite. Note that the profiling is done once and thereafter the re-compiled code is frozen. Unfortunately, improving a program's *average* performance on *typical* inputs does not ensure *optimal* performance on any *specific* input. Worse, it is often difficult to ascertain what constitutes a typical input if the program has a large and characteristically diverse input domain. For instance: What constitutes a "typical" circuit for a circuit simulator? What constitutes a "typical" pedigree for a genetics analysis program? What constitutes a "typical" equation for a symbolic algebra program? Indeed, what constitutes a "typical" program for an interpreter or operating system or compiler?

This thesis explores the conjecture that continually profiling and adapting multiple specializations of a program as they are run on actual inputs can provide greater performance improvement at lower cost than the above-mentioned mainstream primarily-static techniques. The interesting challenges to demonstrating this thesis lie in deciding what code to re-compile, how to do it, to what degree and at what cost. The remainder of this proposal addresses these challenges.

Contents

0.1 Abstract . . . 1
0.2 Problem Statement and Goals . . . 2
  0.2.1 My Thesis . . . 2
  0.2.2 Background . . . 2
0.3 Technical Problems to be Addressed . . . 3
  0.3.1 Profiling SCHEME . . . 3
  0.3.2 Focusing Optimizations . . . 3
  0.3.3 Throttling Optimization Costs . . . 4
0.4 Comparison with Traditional Optimizing Compilation . . . 5
0.5 What I Propose to Build . . . 6
0.6 Proposed Experiments to be Conducted . . . 6
  0.6.1 Human Genetic Pedigree Scenario . . . 7
  0.6.2 Circuit Simulator Scenario . . . 8
  0.6.3 The Strategy and the Criterion for Success . . . 9
0.7 The Specifics . . . 11
  0.7.1 A Concrete Scenario . . . 11
  0.7.2 Scenario SCHEME Code . . . 15
  0.7.3 Profiling: The Scenario and Specifics . . . 16
  0.7.4 Dynamically Adaptive Specialization: The Specifics . . . 36
  0.7.5 Final Comments on the Pedigree Example . . . 41
  0.7.6 Problems and Open Issues: The Specifics . . . 41
0.8 Required Facilities and Resources . . . 45
0.9 Expected Time Table . . . 45

A Pedigree Compilation by P. Szolovits (Reprint) 47
A.1 Pedigrees as Formulæ . . . 49
A.2 Pedigree Formulæ as Computer Programs . . . 52
A.3 Discussion . . . 55

B Pedigree Profiling SCHEME Code 59

C Prototype Program Profiler SCHEME Code 65

[This page intentionally left very nearly blank.]

DESCARTES: A Dynamically Adaptive Compiler and Run-Time System using Continual Profile-Driven Program Multi-Specialization

Doctoral Dissertation Proposal

Michael R. Blair

0.1 Abstract

Continually profiling and adapting multiple specializations of a program as they are run on actual inputs can provide greater performance improvement at lower cost than traditional mainstream compile-time code optimization techniques.

I propose to demonstrate this claim by integrating a traditional optimizing compiler with a code profiler then augmenting the run-time system to use these profiling hooks for adaptive code optimization, using a cost-benefit investment strategy to govern the optimization efforts. Thus, focus for the optimizations is provided by profiling while throttling is provided by the cost-benefit investment model.

Why do I expect this to out-perform traditional compile-time optimization techniques? Traditional techniques for improving program performance rely on information deducible at compile time. Such information is static. It is used to improve the performance of a procedure on its entire domain of possible inputs. Recently developed compiler techniques have yielded further improvements by specializing multiple instances of a program for specific distinct inputs based on textually apparent invocations of the program from within other programs. This too uses only static information. The current state of the art in performance enhancement suggests that gathering information from sample runs of a program (a.k.a. "profiling") can provide still more valuable information. Such strategies gather dynamic information about a program's performance on a suite of typical inputs, re-compiling the program in light of this previously unavailable dynamic information to further improve the expected average performance of the program by improving its performance on the chosen input suite. Note that the profiling is done once and thereafter the re-compiled code is frozen. Unfortunately, improving a program's *average* performance on *typical* inputs does not ensure *optimal* performance on any *specific* input. Worse, it is often difficult to ascertain what constitutes a typical input if the program has a large and characteristically diverse input domain. For instance: What constitutes a "typical" circuit for a circuit simulator? What constitutes a "typical" pedigree for a genetics analysis program? What constitutes a "typical" equation for a symbolic algebra program? Indeed, what constitutes a "typical" program for an interpreter or operating system or compiler?

This thesis explores the conjecture that continually profiling and adapting multiple specializations of a program as they are run on actual inputs can provide greater performance improvement at lower cost than the above-mentioned mainstream primarily-static techniques. The interesting challenges to demonstrating this thesis lie in deciding what code to re-compile, how to do it, to what degree and at what cost. The remainder of this proposal addresses these challenges.

0.2 Problem Statement and Goals

0.2.1 My Thesis

I propose to demonstrate the efficacy of adaptive run-time code optimization guided by continual run-time profiling of object code performance. In short, I shall blur the distinction between compilation time and running time by deferring code optimizations traditionally associated with compile time until a continual run-time profile of code performance justifies the optimization.

This presupposes an economic model of dynamic optimization that answers the questions of what to monitor and how to invest optimization resources. I propose to develop a cost/benefit-based investment strategy. This further presupposes a cost effective run-time profiling facility and mechanisms for dynamic code generation and linking. The later need is satisfied by a growing variety of modern programming languages. Among them are STANDARD ML [Milner 84], C++ [Stroustrup 86], DYLAN [Apple 92], HASKELL [Hudak, Wadler, *et al.* 88], and the entire family of LISP dialects. Of these choices, I propose to use the SCHEME [Clinger & Rees 91] dialect of LISP, primarily because it is a minimalist language in which the important ideas of the project can be readily explored but also because the Mathematics and Computation group in which the research will be conducted (Project MAC) has a high-quality in-house public-domain optimizing compiler (called LIAR) along with the local expertise in its design and implementation. The former concern, effective profiling, has been explored by MacQueen and Appel [Appel, Duba & MacQueen 88] in the setting of STANDARD ML. Their findings readily port to SCHEME and can be incorporated into the cost/benefit model to avoid exorbitant expense— their estimated 40% overhead for profiling is exorbitant in my view.

It is important to emphasize that my findings should readily apply to a variety of modern programming environments since I do not anticipate needing to exploit features of SCHEME which are not also present in the aforementioned other modern languages. Although this proposal may at times appear frightfully SCHEME specific, I have allowed this only to present the research agenda in terms of a very concrete language and its implementation. The final dissertation will endeavor whenever reasonably possible to present the results and conclusions in this more general context. Perhaps, for the final dissertation, a deceptively re-syntaxed SCHEME dialect may suffice in obviating the generality of the ideas.

0.2.2 Background

The main thrust of the thesis is to demonstrate that substantial performance improvements in highly abstractly written scientific code can be automatically obtained by economically judicious, profile-guided adaptive code optimization. I choose abstractly written scientific code as a backdrop because I find it personally interesting, but the techniques I develop will not be limited to that setting: they are generally applicable. The optimizations we consider focus primarily on on-the-fly multiple specializations of time-critical code sections through partial evaluation of the program with respect to its input data.

Recent results in the fields of partial evaluation and abstract interpretation have shown the potential for dramatic performance improvements derived from code specialization (See papers by Berlin and Ungar), but these results have been obtained through requiring the programmer to explicitly specify where specialization is to be done and with respect to what data. There are (at least) three problems with this requirement. First, this requires the programmer to make accurate predictions of where the code will best benefit from specialization, which requires deep understanding of the performance bottlenecks of the program under consideration. This, in turn, often requires careful profiling of the unspecialized code to determine these bottlenecks. Second, this profiling and manual specialization require presentation to the compiler of typical data inputs with respect to which the profiling and specialization should be conducted in order to improve the average performance of the code on such typical data. Finally, these specializations can be prohibitively expensive to perform if done naively. In the worst case, on-line specialization may even fail to terminate.

In sum, optimization through specialization as it is currently portrayed in the field, is an enticing approach to generation of high performance object code but it is at the same time a labor-intensive, imprecise, and often prohibitively expensive endeavor. I propose to soothe each of these problems in turn. First, by automating the specialization based on continual profiling, the process is made transparent to the programmer, eliminating labor by either the programmer or the program user. Further, by basing the specializations on dynamically collected profiling information of actual inputs as they run, the specializations can be made arbitrarily precise with respect to the actual code execution. Finally, the decision of what degree of specialization to perform on what sections of code can be mediated by an economic cost/benefit investment model, regulating the expense of specialization to acceptable bounds. This acts as a throttling mechanism on the optimization process to focus its search through the space of possible optimizations.

0.3 Technical Problems to be Addressed

The technical challenges involved in this research are three:

0.3.1 Profiling Scheme

First, a cost-effective performance monitoring (a.k.a. profiling) mechanism will have to be implemented for the Scheme language since the profiler is the backbone of the specialization decision making process. Fortunately, the efficacious profiling of a very similar language (namely, Standard ML) has already been explored [Appel, Duba & MacQueen 88]. A preliminary discussion with a resident Scheme implementor (Bill Rozas) encourages me to believe that this is essentially a solved problem. Moreover, should the overhead of this approach prove disappointing, another recent paper [Ball & Larus 92] should surely improve the placement of profiling code, though the collected profiling data would require more processing to be actually useful. Most likely, a blend of the two approaches will be adopted.

0.3.2 Focusing Optimizations

Second, various mechanisms for using this profiling information to focus the efforts of a dynamic code optimizer need to be explored. Preliminary work on this has been done by many researchers in the field, among them are [Hansen 74] [Hansen & Wulf 76] [Johnston 79].

Viewing code optimization as a task of balancing the dynamic code optimization efforts against the demands and resources of the "real" computation being performed, dynamic load balancing techniques look very promising by analogy [Kyrimis & Alonso 88]. Most promising is current work at Xerox PARC [Lumer 91]. In short, the optimization resources should initially focus on those code blocks which the profiling data reveals to be performance bottlenecks. Allocating subsequent optimization efforts based on what code initiates the calls to these bottlenecks allows the specialization to gradually diffuse until the root cause of the bottleneck is eventually ascertained.

0.3.3 Throttling Optimization Costs

Third and finally, an economic investment strategy for actually performing the profiling and optimization will need to be developed. Devising a workable investment model should prove the bulk of the work in this thesis.

Specifically, based on the overhead of profiling code, probably not all code should be monitored all of the time. Only code which is likely to prove a bottleneck (such as deeply nested loops) should be monitored. Once a code fragment has been aggressively optimized, further profiling of that code is unlikely to be fruitful: it is more likely to be a distracting tease of where more optimization is desirable albeit beyond the capabilities of the optimization system.

Further questions include: Once bottlenecks have been ascertained, how much effort should be invested into optimization? Is the projected specialization an inexpensive task or will it involve specialization of large traces of code? Does the cost of this specialization justify on-the-fly specialization in a given circumstance or should off-line specialization be performed instead after the computation has completed (like during idle evening hours)?

Would a given abstract and generally useful code fragment be better served by multiple specializations when used in multiple distinct situations? The Stanford SELF project has explored this issue for method specialization in an object-oriented language but we wish to generalize this question to arbitrary higher-order or highly abstract code segments. How should multiple specializations be managed to avoid code explosion due to an arbitrary number of specializations? This has been preliminarily explored in the context of type specialization [Ruf & Weise 91], but we are concerned here with data specialization as well. For example, the values of a matrix may change over many calls to a smoothing function but the shape of the data structure will not. Geneticists are often interested in calculations over a pedigree structure which may have an arbitrarily deep recursive structure [Cooper 90], but any particular pedigree data structure will be fixed in shape and depth. It would be very beneficial to be able to automatically recognize this and specialize the pedigree code to this static data shape [Appendix A].

When multiple specializations are maintained, should their number be dynamically regulated according to some preconceived trade-off between the space they occupy and the performance benefit their presence could yield? When reducing the number of specializations, should they be somehow merged (à la [Ruf & Weise 91]) or should some simply be discarded? Some encouraging results have been developed along these lines in the setting of memory system performance improvement [Baier & Sager 76] [Ryder 74] [Pettis & Hansen 90]. These results should extend to maintaining multiple code specializations.

Nevertheless, producing an economically viable cost control throttling mechanism that does not altogether stifle the investigations of the profiler and optimizer will likely prove a fortuitous task given the delicate balance between the competing objectives. I therefore plan, in the early stages of the thesis research, to not dwell on the costs (within reason) until enough data and experience have been gleaned to direct the throttling. Should the utopian goal of a peaceful and constructive coexistence between the optimizer and the cost throttler prove unattainable within the operating constraints of modern computers, such as they are, the thesis research should nonetheless produce valuable solid new understanding in how the present engineering limitations on hardware and software systems would have to develop in order to make truly continual adaptive optimization desirable. I have no doubt that at some point along the overhead v. gain curve that continually adaptive optimization will be beneficial. The only question is if this occurs only when the profiler and optimizer are so heavily throttled that they, in practice, seldom cross their activation threshold for practical programs of the CPU and memory intensity typical of modern applications. It would be bearable if the answer is that we must wait for computing power (and hence typical application program needs) to go up by a factor of 5 or 10 since this is likely to be witnessed in say the next decade. If we must wait more than a decade for our computers to catch up with this approach, then history suggests that it is unlikely to be adopted by the luminaries in the field at the time it becomes practical. Language system engineering, and indeed most of computer science, seems to be still such a young and fast-changing field that few of the findings that are not immediately applicable today are even remembered tomorrow.

At the very least, anyway, this research should produce a solid foundation for understanding the economic barriers that separate cost-effective compile-time optimizations from cost-effective run-time optimizations. In short, then, the principal research agenda in this proposal is to explore and chart the apparent rift between static and dynamic program optimization so we can build systems that try to take advantage of the benefits of both techniques. That being an awfully vague and lofty goal, I propose to investigate this area within the confines of the far less lofty and much more concrete setting of maintaining continually adapting, multiple specializations of procedures based on profile-guided cost/benefit performance enhancement analysis. I am confident that a practical system that continually adapts can be built by modestly extending contemporary compilation techniques, but there is the ever-present danger that the system may have to mull over a long program run for so long before substantially improving its performance that the benefits are unlikely to be noticed much in practice. Nevertheless, the feasibility of the approach will have been demonstrated and the insight into its practicality will have been thoughtfully explored.

0.4 Comparison with Traditional Optimizing Compilation

Much of the preceding presumes being able to estimate how much an optimization will cost to perform and how much improvement it should stand to yield. Surveying the comparative economics of traditional code optimizations would be a fascinating and valuable project, but probably well beyond the scope of this thesis. It has also at least been tentatively addressed by Craig Chambers' recent Stanford dissertation (as yet unpublished but ftp-able from `self.stanford.edu`). Nonetheless, it would be worthwhile to explore modestly if a gradual deepening strategy of progressive optimization might prove beneficial.

Specifically, it would be nice to tackle one final issue of traditional highly optimizing compilers: namely, why most people do not use them. It became apparent during the MIT Compiler Seminar of Spring 1992, an informal reading seminar composed of researchers and implementors of optimizing compilers, that many of the most aggressive code optimization algorithms are not often used or implemented in production compilers. It was argued that this is for essentially economic reasons. Among them: 1) aggressive code optimizations slow down the compiler too much, 2) optimizations are performed on all of the code whereas it is well established that most of the code execution time will be spend in relatively little of the code, so the time spent in much of the optimization is wasted effort, 3) few companies have the time to dedicate to a final fully optimizing pass of the compiler once their code is ready for release, 4) few companies can dedicate the time and resources to re-testing and verifying the resulting fully optimized code to catch bugs in their aggressive optimization algorithms. I would like to believe that by creating a system that gradually optimizes code that at least the first three of these concerns will have been addressed.

As for testability, this last concern has been at least partially addressed by others [Keppel, Eggers & Henry 91]. It is interesting to note that their work also goes a long way towards making a case for the viability of precisely the sort of dynamically adaptive optimization I propose, although they, like others in the field (namely, Ungar, Hölzle, and Hansen) have remained silent on the possibility of dynamically managing multiple specializations of procedures based on input data.

0.5 What I Propose to Build

I proposal to design and implement a programming system which augments the present MIT Scheme compiler (LIAR) and run-time system [Hanson 91] with the above discussed functionality, as briefly outlined in the previous section.

This will involve modest changes to the LIAR Scheme compiler consisting primarily of installing hooks and access handles into various intermediate phases of the compilation process as well as encapsulating and exposing sundry internal compiler-generated data structures (*e.g.,* , the program graph).

It will also involve augmentation of the run-time system to perform the profiling of Scheme programs as they execute and to coordinate and manage the incremental recompilation process based on an economic model of code optimization.

Finally, I will assess to what degree the thesis has been demonstrated, specifically, that adaptive profile-guided re-compilation is a viable approach to achieve efficient and effective dynamic program performance optimization.

0.6 Proposed Experiments to be Conducted

Following are two programming settings in which I propose to conduct experiments to determine the efficacy of the dynamically adaptive approach to code optimization.

0.6.1 Human Genetic Pedigree Scenario

In short, the drawbacks of partial evaluation boil down to this: multiple specializations of procedures is desirable for performance tuning with respect to the calling procedure and the static (or otherwise stable) components of the procedure inputs. Unfortunately unbridled unfolding of all possible procedures for all known callers and inputs leads to exponential code explosion, and a consequent exponential explosion in compiling time. Traditional and contemporary approaches to dealing with this problem have involved either heuristic throttling of the partial evaluation process based on compile-time bookkeeping and programmer hints or they have involved restriction of the partial evaluation process to only those subdomains of acceptable input programs which are highly unlikely to demonstrate this explosion property in practice. Recent results which demonstrate both of these compromises include: [Berlin 90] [Berlin 89] [Berlin & Weise 89] [Ellis 86]].

My approach is to use the information gleaned from profiling to direct this specialization process toward optimizing only that 10% or so of the code which accounts for the majority of the running time anyway and, in those cases, to perform far more aggressive optimizations than the traditional approaches. Of course, the initial compilation can produce conservative, heuristic-guided object code (not unlike light-weight traditional approaches) but excessive effort should not be invested in heuristic optimizations which may prove inconsequential to the code's actual running time.

As a concrete example of the shortcomings of the traditional approach of static program specialization, consider the problem of computing genetic pedigrees [Appendix A]. Here the programmer presents as input to the pedigree massaging program an instance of a tree-structured representation of a person's family tree with known genetic traits of each tree member. This input is an instance of a recursive aggregate data type of dynamically determinable shape. The pedigree program then iterates some thousand or so times on this tree, repeatedly accessing various properties of the tree elements in a fairly uniform, and hence predictable, fashion.

Without some degree of specialization resulting from partial evaluation of the pedigree program with respect to its input, this redundant and costly structure traversal cannot be avoided without severely restructuring the natural algorithmic expression of the computation [Roylance 87/88]. Thus, either the implementor of the pedigree must distort the pedigree code to suit the inadequacies of the computing environment or the user of the program must distort the data presented so that the pedigree program can be explicitly informed as to what elements of the pedigree will remain static during the program execution. Neither alternative is acceptable, even when the programmer and user are one in the same.

If the user is willing to deal more directly with the computing environment, then an alternative exists: a request can be made of the compiler/run-time system to generate a specialized version of the pedigree program for an anticipated set of commonly structured inputs. Thus, the user viewed as an experimenter can inform the system beforehand of the common structure in a set of data to be processed. However, such programmer hint mechanisms are generally quite simple [Berlin 89] [Berlin 90] [Launchbury 91]. For instance, usually an aggregate input would be represented as an instance of the structure with type names substituted where real component values would normally go and with a special "unknown" marker for data points which will vary structurally among the experiment data. This is an "all or nothing" proposition in that it does not allow even the most primitive composition of sample data, such as saying that component x will be fixed for the first few runs then thereafter component y will be fixed instead. The user must either generate a single specialization in which both may vary or two specializations must be explicitly requested. For fairly structured experiments, this burden on the programmer could be annoying.

More importantly, this effort may in fact be in vain. Without knowing the full implication on program performance with respect to specific input parameters, the user may needlessly labor over minutiæ of fixed versus un-fixed components all to no real avail if performance of the program is relatively unaffected for certain parameters. Thus, to effectively make use of the specialization request facility requires a deep understanding of the specialization process, specifically regarding what specializations it would perform on a piece of code and how those specializations would impact the code's performance. This is just the sort of tedious attention to detail that computers are far better suited to perform than people.

I propose instead to take the user entirely out of the loop, perhaps tolerating user hints or specific requests for specific specializations, but by no means requiring it as the above approach does. Specifically, it should be achievable that running the unadvised adaptive profile-guided run-time system on a pedigree experiment (a series of inputs of common structure and with many common subcomponents) should experience only a modest performance degradation over the duration of an experiment as compared with the performance of a small set of user-requested specializations over the same experiment. Notice that I would include in performance measurement of the latter run the time taken to perform the user-requested specialization of the code in addition to the time of the actual code run. (More on this point later.)

Of course, performing the measurement and comparison of the manual specialization approach against my proposed adaptive approach assumes that some traditional-style partial evaluating compiler for the pedigree program will be available. If not, comparison will be difficult since measuring the performance of a traditional manual approach may require translation of the program into a different language and compilation with a different compiler and execution in a different run-time environment. I anticipate that perhaps some aggressive generalization of the specialization technique described by Berlin [Berlin 89] should suffice to fairly approximate the characteristics of a traditional-style partial evaluator. If nothing else, a "full-manual" mode could be provided for the system I design. This mode would accept user-requested simple specialization guidelines (as briefly described above) and perform complete heuristic static specialization according to standard practice, disabling all profiling and profile-guided adaptation in the run-time system. The result should be a fair approximation of what a contemporary partial evaluating compiler would produce.

0.6.2 Circuit Simulator Scenario

The pedigree scenario above is expected to demonstrate the efficacy of the technique of adaptive profile-driven generation of multiple specializations in the setting of symbolic computation on recursively structured aggregate data where the computation should demonstrate a fairly regular calling pattern. It is expected that my approach will be respectably successful in this setting.

To balance the assessment of the efficacy of this technique, however, I further propose conducting an experiment in the setting in which traditional techniques seem to perform fairly well. Specifically, unlike symbolic programs, "scientific " programs involving fixed structures with time-varying component values will be conducted. The most opportune such scenario involves direct comparison between the performance of my approach and that of Berlin [Berlin 89] in the context of a digital circuit simulator.

Specifically, I propose to use Berlin's augmented SCHEME compiler and run-time system to compile a series of characteristic circuits and collect the total time and space requirements for the compilation of the circuits and the time and space used in simulating them for some number of iterations. I will then measure the same parameters with the same circuits using the dynamically adaptive specialization system. I hope to characterize the measured differences to assess to what extent my approach might be preferred to the traditional approach.

0.6.3 The Strategy and the Criterion for Success

So, how do I substantiate the claims that a profile-guided adaptive approach can outperform more traditional specialization strategies? Let me address that by first qualifying under what circumstances I expect the dynamically adaptive technique to prevail and what criteria I use to decide what it means to prevail.

The metrics of interest. First, in measuring performance I will be interested in both time and space performance. Although I consider time to be the more important of the two metrics (we can always buy more memory but it's not so easy to extend your life), I will nonetheless pay heed to space utilization to verify that: a) I'm not cheating by making outrageous numbers of specializations (like one for each call site ever encountered), and b) I'm not winning or losing due to mere paging behavior alone. I anticipate collecting timing measurements then cross-correlating them with the space utilization as a reality check.

How the measurements will be performed. Second, in the interest of fairness, when measuring time and space of my approach I will be measuring time and space of both the actual run of the code and of the profiling and incrementally adaptive re-compiling processes. Since this is the resource utilization the user will actually witness when using my system, it is the honest way to measure my system's performance. To be fair, therefore, when measuring the performance of traditional approaches I will likewise include the time and space of the compilation process as well as that of the run of the code. Of course, there is one other sticky issue in how I measure my system's performance which might give me an unfair advantage: my system is continually adaptive over the entire lifetime of the object code. Thus, if three distinct experiments, each consisting of several top-level invocations of the program, are run, I think it only fair to distinguish between a) recompiling the program and starting the run afresh for each distinct experiment and b) allowing the code to persist so that future experiments might benefit from adaptations gleaned from prior experiments. I feel it will be important to measure these performance traits separately since the traditional user who is compelled to manually re-specialize upon each new experiment will experience that compiling time cost (not to mention the off-line mental overhead of deciding how to do the specialization and typing in the actual specialization command to the machine). The user of my system, however, should not be compelled to perform such manual housekeeping chores.

Great expectations. Now, as for timing comparisons, I don't realistically expect to consistently out-perform traditional compiler techniques. Specifically, I anticipate some number of cross-over points between performance curves of the two approaches. For instance, for a small number of iterations, I should always win because the compilation time of the traditional approach will dominate the run. For larger numbers of iterations I may lose for a while because I continually profile and adjust, which entails some non-negligible amount of overhead. Past some much larger boundary, however, once the code has stabilized to the point that most of the profiling is suspended, I should again prevail because my system will have more aggressively fine-tuned the code to the specific input data set being manipulated. The entire premise of specialization via partial evaluation is that finding the stable components of the input and of the program's control structure enables dramatic performance improvement in the resulting specialized program. Since my approach acquires dynamic information through profiling— information that is statically unavailable— I can incorporate knowledge into the specialization that traditional techniques cannot. Moreover, I can progressively strengthen heuristic sub-decisions in the specialization process as more evidence becomes available that further specialization would be beneficial and cost-effective.

So where *specifically* do I anticipate I will gain this expected performance advantage. Aside from the mantra that I don't waste time optimizing code that doesn't get run (much), I gain most of my advantage by being able to very aggressively specialize those code fragments that does account for a majority of the running time. Specifically, a classic shortcoming of on-line compile-time partial evaluation strategies is that they steadfastly avoid unfolding procedure calls across "non-trivial" conditionals, namely those conditionals which reflect conditional transfer of control between code blocks. To do otherwise would be to risk non-termination in the partial evaluator, and hence in the compiler, as is now well known in the partial evaluation community. The root of the problem is that unfolding across conditionals, once embarked upon, is very difficult to throttle in any reasonable way other than to statically limit the unrolling depth [Ellis 86] or some such similarly trivial heuristic termination criteria. Most just avoid it altogether.

Two nice examples of this concern are sketched by Adams [Adams 90]. The problem of unrolling across conditionals is more carefully developed in the context of trace scheduling (which is tantamount to an intermediate instruction level partial evaluation) by Ellis [Ellis 86]. Finally, the formal underpinnings of this termination concern are developed fully by Launchbury [Launchbury 91].

I anticipate that a profile-guided dynamically adaptive approach may prevail even in the setting of scientific code where traditional approaches perform quite well. I believe this principally because I will be able to specialize across conditionals, which traditional strategies cannot yet do. Of course, that is a vigorously pursued open problem in partial evaluation, but one which is unlikely to give the traditional approach the edge against my approach. This is principally because my approach can focus its optimization efforts on the most crucial bottlenecks in a way that dominates the static heuristics employed by traditional methods. The bottom line is this: traditional approaches spend time optimizing code which may not prove performance-critical and traditional approaches must rely on heuristic focus and direction which falls shy of aggressively optimizing what may prove the most critical bottlenecks. My approach, by contrast, invests, at run time, in ferreting out the true bottlenecks and in aggressively optimizing them as focused and directed by dynamically revealed information provided by the code profiler. I can unroll across precisely those conditions which are performance critical and do so only to the extent that the profile information deems it fruitful. This gives me precise focus and assured termination.

An intriguing final set of experiments. Finally, given that the compiler hooks and run-time environment augmentations will be incorporated into a Scheme compiler, coupled with the fact that the compiler itself and much of the run-time environment are themselves written in Scheme, I look forward to the experiment of assessing to what extent the compiler and run-time system could benefit from their own augmentations. That is, recompiling the compiler and run-time system with the dynamically adaptive extensions should characteristically alter their performance, and hence the performance of the overall system. Collecting separate sets of measurements for before and after experiments of re-compiling the system with adaptive extensions could prove interesting if not downright exciting. It is not immediately clear to me whether these second order effects could serve to stabilize or destabilize the profiling and optimizations attempted. I am very curious to observe its behavior directly and characterize what I witness.

0.7 The Specifics

Let's consider a concrete setting in which these vague techniques can be made more precise. This exercise should clarify many questions the critical reader will no-doubt have formulated. To wit, what specifically will change in the present compiler and run-time environments and how will they change? In other words, what are the pieces of this strategy and how do they fit together? To address these concerns, I consider first the augmentations to support profiling. Afterward, I address the more central issue of dynamically adaptive multiple specializations.

0.7.1 A Concrete Scenario

I present the specifics within the context of the following concrete programming example, as presented in [Appendix A]. This is the example, alluded to earlier, of computing probabilistic human genetic pedigrees from recursively tree-structured family pedigrees. No understanding of genetics or pedigree counselling is assumed or necessary to penetrate the essential aspects of this example. A fairly casual read through [Appendix A] would more than suffice to set the stage for the details of what I am proposing to build. For the impatient reader, the next few paragraphs serve as a quick overview of the contents of that appendix. For the more thorough reader, these next few paragraphs should set the stage for how I offer to address some of the problems raised in that appendix article.

The pedigree input data structure

The user of the pedigree program provides, as input, a tree structure like the one below (Figure 0-1 (b)). Some of the nodes of the tree contain known static values (the shaded ones of (b)) while others are of unknown value that range over finite domains.

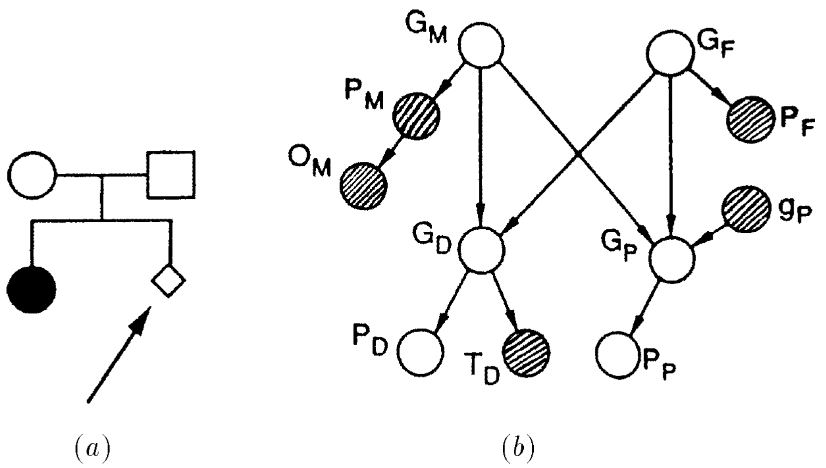


Figure 0-1: A Sample Family Pedigree Tree – (a) A very simple pedigree, as it might arise in genetic counseling. The couple have had a daughter affected by a suspected Mendelian disorder whose mode of inheritance is not certainly known, and are concerned about the risk to the propositus, who is indicated by the arrow. (b) The pedigree of (a) corresponds to a network of variables interrelated by probabilistic dependencies. M, F, D, and P subscripts on nodes correspond to the mother, father, daughter, and propositus, respectively. Nodes labeled G stand for the individual's genotype, P for phenotype, g for gender, T for a test whose outcome depends on the genotype, and O for an observation whose outcome depends on the phenotype.

Specifically, in a LISP implementation of the pedigree tree, one might represent known values as scalars while representing unknown values as vectors (LISP arrays) over the possible value domain of the parameter. The tree might be represented as a linked list structure, each node being a list containing the value of the node and pointers to the node's successor nodes.

The equation for probable pedigree expression

In the terminology of genetics, the probability that a pedigree configuration will occur in nature is dubbed the "probability of expression" of the pedigree, that is, the probability that the pedigree will be "expressed" in nature. The task of the pedigree program is to assess the probability of the tree as a whole being expressed when constrained to observe the fixed (known) values. This is computed by iterating over the possible values of the unknown nodes, computing the conditional probability of the tree as a whole being expressed. These per-node conditional probabilities, in turn, are computed as *a priori* probabilities of occurrence for root nodes while the probabilities for internal and leaf nodes are computed according to their Bayesian conditional probabilities based on the values (known or conjectured) of their ancestor nodes. These *a priori* and conditional probabilities are usually tabulated according to observed probabilities in a large sample population. In general, this computation of the probability of expression of a pedigree tree as a whole can involve an exponential number of per-node probability assessments, each of which can involve several multiplications and additions, usually over floating-point values.[1] This can be prohibitively expensive to compute.

For example, the fully general equation for the probability of expression of a pedigree of the shape of Figure 0-1 is shown in Figure 0-2 (adapted from Equation 3 of [Appendix A]):

$$
\begin{aligned}
P(t) = \sum_{P_M}\sum_{G_D}\{ & \sum_{G_F}[\; P(G_F) \times \\
& \qquad \sum_{g_P}(\; P(g_P) \times \\
& \qquad\qquad \sum_{P_P}\sum_{G_P}\{\; P(P_P \mid G_P) \times \\
& \qquad\qquad\qquad \sum_{G_M}[\; P(G_P \mid g_P, G_M, G_F) \times \\
& \qquad\qquad\qquad\qquad P(G_D \mid G_M, G_F) \times \\
& \qquad\qquad\qquad\qquad P(P_M \mid G_M) \times \\
& \qquad\qquad\qquad\qquad P(G_M)\,]\})\times \\
& \qquad \sum_{P_F} P(P_F \mid G_F)\,] \times \\
& \sum_{T_D} P(T_D \mid G_D) \times \\
& \sum_{P_D} P(P_D \mid G_D) \times \\
& \sum_{O_M} P(O_M \mid P_M)\,\}
\end{aligned}
$$

Figure 0-2: The fully general equation for the probability of expression of the pedigree tree t depicted in Figure 0-1(b).

To understand how this equation was derived (which is not essential to the remainder of the discussion), first imagine that each subscripted summation symbol ($\sum$) were pulled out to the very beginning of the overall equation. This does not alter the value ultimately computed: it just introduces redundant multiplications into the evaluation of the equation when the equation is viewed as a procedural prescription for how to compute $P(t)$. It then becomes more apparent that the equation is summing over all attainable configurations of node values, computing the conjunctive probability (product) that each node has attained its present value given the value(s) of its immediate predecessor node(s). Specifically, each $P(\ldots)$ term is the probability of a node attaining its present value. For example, $P(g_P)$ is the *a priori* probability of the gender of the propositus attaining the present value of the parameter g_P; $P(G_D \mid G_M, G_F)$ is the conditional probability of the genotype of the daughter attaining the value G_D given that the genotype of the mother and father are G_M and G_F respectively. In general, the probability of attaining the value of a given parameter

[1]Assuming that *a priori* probabilities are expressed in the traditional value range $[0, 1]$.

is the conditional probability of the corresponding node in the pedigree tree attaining that parameter's present value given the present values of all of its immediate predecessor nodes in the pedigree tree. The particular ordering of the $P(\ldots)$ subterms and the subsequent pushing inward of the summation operations ($\sum$'s) in the given equation was chosen solely on the grounds of avoiding redundant multiplications.

For the extremely simple sample tree given, the not-atypical configuration of known and unknown values depicted by the shading of known values in Figure 0-1 (b) would yield the specialized equation of Figure 0-3 [Appendix A]:

$$
\begin{aligned}
P(t) = \sum_{G_D}\Big\{ \sum_{G_F}\Big[\; & P(G_F) \times \\
& P(GP) \times \\
& \sum_{P_P}\sum_{G_P}\Big\{\; P(P_P \mid G_P) \times \\
& \qquad \sum_{G_M}\Big[\; P(G_P \mid GP, G_M, G_F) \times \\
& \qquad\qquad P(G_D \mid G_M, G_F) \times \\
& \qquad\qquad P(PM \mid G_M) \times \\
& \qquad\qquad P(G_M) \;\Big]\Big\} \times \\
& P(PF \mid G_F) \;\Big] \times \\
& P(TD \mid G_D) \times \\
& P(PD \mid G_D) \times \\
& P(OM \mid PM) \;\Big\}
\end{aligned}
$$

Figure 0-3: A typical pedigree computation. The non-subscripted parameters are "known" (*viz.,* GP PM PF TD PD OM PM) whereas the subscripted parameters (which appear themselves as subscripts of various summation operators) range over a finite set of possible values with known *a priori* probabilities of expression.

This was derived from the previous equation by simply dropping the (trivial) summations over known parameters and renaming those known parameters by raising their subscript to make two-letter parameter names (following Szolovits).

Discussion

Implementing a program to efficiently derive and solve a equation of this form gives rise to two central problems:

Problem 1: Exploiting Stability of Structured Data Note first of all that this equation is not given directly to the program. Rather, the program must deduce this equation from the structure of the input supplied (*viz.,* the tree of Figure 0-1 (b)). As alluded to above, a pedigree tree would be implemented as a tree-structured datum which is an instance of a recursive aggregate type (such as a LISP list structure). Therefore, the precise shape of the input data cannot even be statically determined. As a result, a naive implementation of the pedigree program might spend much of its time needlessly re-traversing the tree structured datum.

To deal with this problem, the program for computing the probability of pedigree expression of an arbitrary pedigree tree might naturally proceed in two phases: first walking over the recursively structured input tree to deduce the appropriate equation, then in phase two evaluating the value of the deduced equation. This task of deriving the desired equation from the recursively structured input datum Szolovits characterizes as a preliminary interpretive phase of the pedigree computation. The second phase he characterizes as more of a strictly computational phase, in the traditional arithmetic sense.

Problem 2: Exploiting Stability of Control Second, note that the derived equation involves nested loops (the innermost summation being the innermost loop) that iterate over a fixed range of values for each variable node of the tree structure, thus giving rise to a regular (and hence predictable) pattern of node accesses and arithmetic sub-computations. Some of this regularity is exploited by rearranging the subterms of the summation then moving the summation operators inward when possible, as described in the informal derivation following the equation of Figure 0-2.

Szolovits argues that this second concern cries out for specialization. Imagine, for instance, that this very simple tree structure were in fact expanded to encompass several generations of a large family, each child node bearing a few offspring. Szolovits demonstrates that through common compiler optimization techniques (such as constant folding and loop unrolling) that the computation can be reduced to a tenable level. For instance, in the sample equation given, exploiting the fact that some parameters remain fixed by moving some summation operators inward results in roughly one third as many multiplications as the fully general equation with all summation operators at the front of the equation. This inward migration of summation operators is, in essence, a memoization of the unmigrated equation. Further, Szolovits found that in-lining the remaining summation operators and unrolling the resulting summations could result in an additional factor of 3 or 8 speedup. Overall, with additional common optimizations (such as constant folding), Szolovits observed speed-ups of as much as an order of magnitude on large pedigrees.

He leaves as an open problem, however, how one is to decide how to perform the optimizations fruitfully in light of the potential exponential blow-up in code size, especially due to loop-unrolling. As he suggests, optimization of the innermost loops is the crucial first step, but how aggressively to optimize them and what decision criteria to use for this optimization remains open.

In the following sections, I propose to demonstrate a technique for tackling precisely these two problems. Specifically, the next several subsections address the issues behind problem 2 above. Afterward several sections addressing problem 1 are presented.

0.7.2 Scenario Scheme Code

For the moment, let's assume that the interesting issue is not in generating Scheme code to massage the pedigree from the general tree-structured input. Specifically, let's imagine that after exploring the tree-structured pedigree of Figure 0-1 and deducing the equation of Figure 0-3, the following Scheme code of Figure 0-4 is generated (which uses the system-provided utility subroutines `vector-summation`, `sc-*`, and `P` of Figure 0-5).

In this code, $\sum$ is implemented as `vector-sum` whereas the implied multiplications are implemented, following Szolovits, as a short-circuit multiply (`sc-*`). This short-circuit multiply takes a variable number of thunks (procedures of no arguments) and evaluates their procedure bodies in turn from left to right (by applying them to no arguments). If at any point a thunk's body evaluates to 0, the product is 0 so no more thunk bodies need be evaluated. In addition, the probability operator P is here implemented as a simple (in fact, trivial) arithmetic function rather than as a lookup into a table of empirically observed conditional probabilities. That is, rather than allowing parameters to vary over symbolic values (such as "homozygous" or "heterozygous" for genotypes), I here make parameters vary over the *a priori* probability of the associated symbolic value. Moreover, I assume all values attainable by a parameter to be equally probable. Neither of these simplifications affects the characteristic behavior of the pedigree program with respect to the program presented by Szolovits.

Finally, note the commented labels at each call site of `vector-summation`, `sc-*`, and `P`. These are indexed by their subexpression nesting depth and index within the subexpression in which they occur. For instance, `v-`$\sum$`-1.0` indicates a call site of `vector-sum` which is at depth 1 (the expression which is the body of `P_pedigree`) and it appears at index 0 (the operator of the given subexpression). `P-4.1`, by contrast, is nested 4 levels deep and appears at index 1 (the first argument in the enclosing subexpression).

To be very concrete, with the *a priori* parameter values in Figure 0-6, the call of `(P_pedigree)` returns 0.015625, or a $1\frac{9}{16}\%$ probability of this pedigree being expressed.

The remaining subsections illustrate the dynamically adaptive optimization technique I propose to implement using the above code examples. Specifically, the next subsection introduces the toolkit we will use to isolate bottlenecks in our code and demonstrates precisely how these methods apply to our specific pedigree program example. The major next section discussed the particulars of what sort of dynamically adaptive specializations might be attempted and what the issues are surrounds those potential attempts at performance enhancements. the section following that contemplates the potential difficulties and unanswered questions in this approach to code optimization.

0.7.3 Profiling: The Scenario and Specifics

There are essentially three fundamentally distinguishable approaches to program profiling (at least, as nicely outlined by Samples [Samples 91]). The first is what he characterizes as the "ancient" technique of PC sampling. The second is a per-procedure-call registration. The final technique is full-blown tracing of entire execution passes. These techniques, as listed, provide progressively more detailed and more useful information for program optimization. They are also in order of increasing overhead. I propose to implement a system that gradually shifts from one to the other as the focus of the optimization effort sharpens. Perhaps not all phases will ultimately be employed by the final implementation, but for purposes of guiding the research it should prove instructive to initially use all three.

Returning to the pedigree example introduced above, consider how profiling might isolate the places where some program optimization might be beneficial.

```
;;;
;;; Pedigree Sample SCHEME Code
;;;

(define (P_pedigree)
  ;;------------------------------------------------------------------------
  "The codified equation for probability of pedigree expression from Figure~0-3"
  ;;------------------------------------------------------------------------
  (vector-summation G_D                                        ;  v-Σ-1.0
    (lambda (g_d)
      (sc-*                                                    ; sc-*-2.0
       (lambda ()
         (vector-summation G_F                                 ;  v-Σ-3.0
           (lambda (g_f)
             (sc-*                                             ; sc-*-4.0
              (lambda () (P g_f))                              ;    P-4.1
              (lambda () (P gP))                               ;    P-4.2
              (lambda ()
                (vector-summation P_P                          ;  v-Σ-5.0
                 (lambda (p_p)
                   (vector-summation G_P                       ;  v-Σ-6.0
                     (lambda (g_p)
                       (sc-*                                   ; sc-*-7.0
                        (lambda () (P p_p g_p))                ;    P-7.1
                        (lambda ()
                          (vector-summation G_M                ;  v-Σ-8.0
                            (lambda (g_m)
                              (sc-*                            ; sc-*-9.0
                               (lambda () (P g_p gP g_m g_f)) ;    P-9.1
                               (lambda () (P g_d    g_m g_f)) ;    P-9.2
                               (lambda () (P PM     g_m    )) ;    P-9.3
                               (lambda () (P        g_m    )) ;    P-9.4
                               ))))))))))
              (lambda () (P PF g_f))))))                       ;    P-4.4
       (lambda () (P TD g_d))                                  ;    P-1.2
       (lambda () (P PD g_d))                                  ;    P-1.3
       (lambda () (P OM PM))))))                               ;    P-1.4
```

Figure 0-4: SCHEME code to implement the equation from Figure 0-3.

```
;;;
;;; Pedigree Utility Subroutines
;;;

(define (vector-summation v proc)
  ;;---------------------------------------------------------------------------
  "(VECTOR-SUMMATION v proc) sums the invocation of PROC on each vector V elt."
  ;;---------------------------------------------------------------------------
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc             0             (+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))


(define (sc-* . thunks)
  ;;--------------------------------------------------------------------------
  (string-append
   "(SC-* . thunks) short-circuit times of thunk calls from left-to-right."    newline-string
   "If any thunk call yields 0, short-circuits to return 0 w/o calling thunks" newline-string
   "to its right.")
  ;;--------------------------------------------------------------------------
  (do ((thunks-2-do thunks (cdr thunks-2-do))
       (acc           1    (* acc ((car thunks-2-do)))))
      ((or (null? thunks-2-do)
           (zero? acc))
       acc)))


(define (P val . givens)
  ;;-------------------------------------------------------------------------------------
  "(P val . givens) computes the conditional probability of VAL given all GIVENS values."
  ;;-------------------------------------------------------------------------------------
  ;; This would actually do a table lookup in a table of empirically observed
  ;;  conditional probabilities.
  ;; To keep things simple for now, we just effectively return VAL.
  (let ((given-val (reduce * 1 givens)))
    (/ (* val given-val)
       given-val)))
```

Figure 0-5: Pedigree computation utility subroutines.

```
;;;
;;; Sample Pedigree ''Known'' Parameters
;;;

(define G_M (make-vector 4 (/ 1.0 4)))
(define PM .5)
(define OM .5)

(define G_F (make-vector 4 (/ 1.0 4)))
(define PF .5)

(define G_D (make-vector 4 (/ 1.0 4)))
(define PD .5)
(define TD .5)

(define G_P (make-vector 4 (/ 1.0 4)))
(define gP .5)
(define P_P (make-vector 8 (/ 1.0 8)))
```

Figure 0-6: Pedigree sample parameters.

PC Sampling

PC sampling is the practice of making the run-time system periodically process a timer interrupt that samples the value currently in the program counter (PC) of the virtual machine that is running the program. These sampled values are used to develop a histogram of the statistical frequency with which each procedure is activated. For instance, a table associating each procedure with a counter could be maintained. Each time the sampled PC indicates that some user-defined compiled procedure p is the currently active procedure, p's counter is incremented.

To perform PC sampling in MIT Scheme, compiled user-defined code should be loaded into MIT Scheme's constant space. This makes PC sampling immune from garbage-collector interference by making the garbage collector not move the procedures.[2] This PC value can be mapped to a procedure header by using handles currently available in the MIT Scheme trap handler mechanism and compiled-code pretty printer code. Specifically, the trap handler contains code that can map the PC to a code header and the pretty printer contains code that can map a code header and offset to a procedure object. These code fragments can be abstracted out of the respective handlers and made available as Scheme primitives.[3]

For example, a naive (read: expensive) prototype PC sampler produced, for the execution of the pedigree procedure of Figure 0-4, the histogram in Figure 0-7. This PC sampler assumes that the primitive procedures from which the user-defined procedures are composed

[2] Alternatively, one could leave procedures in the garbage collected heap but being sure to process all PC samples into procedure names before any gc motion is executed.

[3] In fact, Bill Rozas has argued that, for other reasons, this should be done anyway.

are all (naturally) in-line coded.[4] [The prototype profiling code is attached as an Appendix to this proposal for completeness]. Here, the proc-names correspond to the labels in the program of Figure 0-4, D re-iterates the nesting depth of each procedure, count indicates the total number of times the PC sampler sampled this procedure as the currently active procedure, and percent indicates what percentage of the time the procedure was detected to be the currently active procedure. The bottom of the histogram lists, as a reality check, the total number of procedure activations detected and the total of all the percentages (which should closely approximate 100).

```
((=========-PC-SAMPLING-========)
 (-- ----------- - ----- -------)
 ( N  proc-name  D count percent)
 (-- ----------- - ----- -------)
 ( 0 ---sc-*-9.0 9    64  43.243)
 ( 1 ---p----9.1 9    16  10.810)
 ( 2 v-sigma-8.0 8    15  10.135)
 ( 3 ---p----9.3 9    14   9.459)
 ( 4 ---sc-*-7.0 7    10   6.756)
 ( 5 ---p----9.4 9    10   6.756)
 ( 6 ---p----9.2 9     9   6.081)
 ( 7 ---p----7.1 7     5   3.378)
 ( 8 v-sigma-6.0 6     4   2.702)
 ( 9 v-sigma-5.0 5     1    .675)
 (10 ---p----1.3 1     0   0    )
 (11 v-sigma-3.0 3     0   0    )
 (12 ---p----4.4 4     0   0    )
 (13 v-sigma-1.0 1     0   0    )
 (14 ---p----0.0 1     0   0    )
 (15 ---p----4.1 4     0   0    )
 (16 ---p----1.4 1     0   0    )
 (17 ---sc-*-4.0 4     0   0    )
 (18 ---p----1.2 1     0   0    )
 (19 ---sc-*-2.0 2     0   0    )
 (20 ---p----4.2 4     0   0    )
 (                 ----- -------)
 (         Totals:   148  99.995))
```

Figure 0-7: Pedigree example PC sampling.

Referring only to the equation of Figure 0-3, one might expect the most deeply nested summation (with its associated $P(\ldots)$ and implicit multiply) to be the most severe bottleneck in the computation. The general trend should be that the more deeply nested calls are sampled more frequently, though perhaps with calls to $P(\ldots)$ not sampled as frequently as other procedure calls at the same depth since $P(\ldots)$ is a fairly trivial (fast) computation

[4] It would not be a disaster if they are not in-line coded. It might just be moderately distracting to the optimizer.

so statistically it should be detected less frequently than the enclosing `sc-*` at the same depth.

This is precisely the general trend observed in Figure 0-7. Some 43% percent of the time is spent in the innermost summation, with the less deeply nested procedures sampled less frequently. Picking an arbitrary attention threshold of, say, 5%, this PC sampling could be interpreted as revealing that some effort should be made to accelerate those procedure calls at depths 8 and 9 as well as the multiplication at depth 7. All remaining procedure call sites are active less than the threshold of interest, so they will not be further optimized at this time.

The next step is to start gathering more information about these apparently critical procedure invocations so that optimization diagnosis can be performed. Specifically, the next phase of profiling is to initiate block tracing.

Block Tracing

Block tracing, as described in detail by [Appel, Duba & MacQueen 88], involves recording the time spent in the basic blocks that comprise the bodies of user-defined out-of-line procedures. In our ongoing scenario, block tracing would be initiated on the basic blocks comprising the procedure call sites which PC sampling revealed as potential bottlenecks. This would be done in order to assess whether some aggressive optimization of these apparent bottlenecks might be fruitful. It is estimated that this can be done at only (say) a 10–20% overhead typically.

It should be noted that the time recorded by block profiling as "time spent in a procedure" is not what one might naturally expect. Specifically, the recorded time is the time measured to have been spent in the indicated procedure call site, *not counting the time spent in out-of-line procedure calls invoked from this call site.* In other words, block profiling really just counts, for a given code block, the time spent for which the PC was within the range of instructions comprising that code block, not the time spent awaiting the final exit from the code block.

At the risk of belaboring this fairly subtle point, consider the following mutually recursive procedures:

```
(define (zero? num)
  (= num 0))

(define (even? whole-num)
  (or (zero? whole-num)
      (odd? (- whole-num 1))))

(define (odd? whole-num)
  (and (not (zero? whole-num))
       (or (= whole-num 1)
           (even? (- whole-num 1)))))
```

Using these definitions, if one were to perform block tracing on the top-level call `(even? 42)`, one might naturally expect that if the evaluation of the result takes time t, then the time spent awaiting the final return of the initial call to `even?` would be t. This, however, is not the time that block tracing would record as time spent in `even?`. Instead, the block trace would count only that time spent in `even?` that is not time spent in `zero?` or in `odd?`.

To appreciate why this is a useful way to record time expenditures, consider the sample block trace for our example pedigree code shown in Figure 0-8. Again, this was generated by a naive prototype profiler. Although I have proposed a strategy that would resort to block tracing only of those procedures revealed, by PC sampling, to be apparent bottlenecks, the table shown nonetheless block traces all user-defined procedures. This I do in order to dramatize the fact that those procedures deemed non-bothersome by the PC sampling are indeed of little concern, at least according to the "weighted" percentage (discussed below) of the block tracing table.

```
((===================-BLOCK-PROFILING-=====================)
 (--- -----------  -  ------ ------ ------------- ---------)
 ( N   proc-name   D    time norm %  weighted time weighted %)
 (--- -----------  -  ------ ------ ------------- ---------)
 ( 0  ---sc-*-9.0  9   9.539  8.505 3695991465.059     80.258)
 ( 1  ---p----9.1  9    .520   .463  201458654.280      4.375)
 ( 2  v-sigma-8.0  8  11.049  9.851  185388236.799      4.026)
 ( 3  ---p----9.2  9    .469   .419  182087629.829      3.954)
 ( 4  ---p----9.3  9    .460   .410  178213424.940      3.870)
 ( 5  ---p----9.4  9    .390   .347  151093990.710      3.281)
 ( 6  ---sc-*-7.0  7  12.269 10.939   10104872.609       .219)
 ( 7  v-sigma-6.0  6  12.739 11.358     594397.439       .013)
 ( 8  ---p----7.1  7    .190   .169     156473.170       .003)
 ( 9  v-sigma-5.0  5  12.859 11.465      40187.499       .001)
 (10  ---sc-*-4.0  4  12.899 11.501       3302.399      0.    )
 (11  v-sigma-3.0  3  12.909 11.510        348.569      0.    )
 (12  ---sc-*-2.0  2  12.919 11.519         51.679      0.    )
 (13  v-sigma-1.0  1  12.939 11.537         12.939      0.    )
 (14  ---p----4.4  4   0.      0.          0.           0.    )
 (15  ---p----0.0  1   0       0           0            0     )
 (16  ---p----4.1  4   0.      0.          0.           0.    )
 (17  ---p----1.3  1   0.      0.          0.           0.    )
 (18  ---p----1.2  1   0.      0.          0.           0.    )
 (19  ---p----4.2  4   0.      0.          0.           0.    )
 (20  ---p----1.4  1   0.      0.          0.           0.    )
 (                   ------- ------ ------------- ---------)
 (           Totals: 112.15  99.993 4605133047.920    100.    ))
```

Figure 0-8: Pedigree example block profile.

In this table, the `time` field shows the time measured as described above. The `norm %` (for "normal percentage") column indicates what percentage of the total time was spent in the block of a given procedure. Re-ordering the table entries by these percentages would not properly reveal the principal bottlenecks given that our intent is simply to treat the distance of some entry from the top of the table as indicating decreasing priority for optimization. Specifically, sorting according to the normal percentage would result in an ordering that roughly reverses the ordering actually shown in the table but with all the traces of $P(\ldots)$ moved to the end of the trace histogram and these $P(\ldots)$ traces left in the relative order in the table shown. Such an ordering would not be as useful as the ordering shown in the table since it would not identify a prioritization of bottlenecks that is focused by nesting depth. On the other hand it would also be misleading to sort the table based first on nesting depth then, within each equal depth subsort, sorting on the normal percentage. This is misleading in that, for example, it would always give priority to a more deeply nested procedure that consumes very little time than to a less deeply nested procedure in which a majority of the overall system time is consumed. Therefore, the standard focusing technique is to somehow weight the measured times by the nesting depth of the procedure call site.

The `weighted time` column incorporate the nesting depth (`D`) of the traced procedure into the empirically measured time. The particular weighting function for this particular profile was D^D. This particular weighting scheme was carefully selected so as to produce a ranking of weighted percentages that strongly resembles the ordering in the PC sampling presented earlier (Figure 0-7). It is not, however, clear that this is a desirable goal in the long run. It merely illustrates that some form of weighting based on nesting level is desirable. Some researchers use a simple linear weighting while others use a more polynomial weighting (like D^2) and even one author uses an exponential weighting (roughly 2^D) [Hansen 74]. So long as it focuses the optimizers attention on the true bottlenecks, the specific weighting equation is not that important. To conclude, the `weighted %` column is just the ratio of each weighted time to the total weighted time, expressed as a percentage.

At any rate, activating block tracing of a compiled procedure is fairly easily done in MIT SCHEME since the procedure call mechanism is structured around an implicit trap at each procedure call. This trap is normally not active at a procedure call, but we can arrange to modify the procedure's header to will make it trap each time it is called. This trap would perform the block trace bookkeeping by recording the elapsed time since the last trap (last block entry) into the block trace histogram entry for the block currently being exited, then restarting the block trace timer, finally returning from the trap and executing the target procedure's body as normal. Upon returning from a call that appears as a component of a sequence of expressions (for instance, processing the argument subexpressions of a multi-argument procedure or processing the subexpressions of a `BEGIN` form), a similar trap would need to be processed to ensure that it is the caller who is charged for subsequent non-out-of-line calls and not the last out-of-line callee. This strategy is very careful not to break tail recursion.

Data-specific Block Tracing

Since we wish to perform input data specific specialization, we could, after assessing where the most prolific bottlenecks are located (say, those entries with a weighted percentage above 5%), augment these block tracing entries by breaking them down into separate sub-entries for each distinguishable pattern of argument values encountered by a call site. By separating these timing profiles based on the input data, memoization/specialization of the bottleneck procedures can be considered. For instance, noting that our apparent bottleneck in our pedigree example is the innermost `sc-*` call, we would likely focus on that as a starting point for input-focused optimization.

More specifically, in the innermost loop of our pedigree example, the `sc-*-9.0` procedure will be called on instances of the the same thunks each time it is called. With each iteration of the enclosing summation, new thunk instances will be created but these procedure objects are immutable. In fact, by knowing a little about SCHEME procedure objects, the system could note that although the environment over which these procedures are closed changes with every iteration, the structure of that environment does not change, nor does the code associated with the procedure objects. By capitalizing on this small amount of knowledge, the run-time system could in-line code this call to `sc-*`, in effect specializing it to the four known thunks on which it is always called. By augmenting the block tracing profiler to record the arguments on which the `sc-*-9.0` textual instance of `sc-*` is invoked (along with the other information normally recording during block tracing), the discovery that these thunk instances share common code and environment structure can be discovered.

Further, argument profiling the calls to `P` appearing in these thunk arguments to `sc-*-9.0` reveals that some arguments remain relatively fixed in value for a long while; specifically, the outermost iteration subscripts remain stable for much of the invocations of the innermost summation. Thus, again, multiple on-the-fly specializations might be in order. Keep in mind that our purpose in these sections on profiling is to detail how the system will detect where the multiple specialization might be performed by demonstrating what sort of information will need to be collected by the profiler(s) and how that information should be presented to the run-time optimizer. The details of exactly what multiple-specializations the optimizer will then actually perform will be discussed in a later section. Nonetheless, we should still look in a bit more detail now at how the system decides where to initiate input-specific block tracing.

In order for the run-time system to decide to perform such detailed tracing of this innermost multiplication and $P(\dots)$ calls, the threshold for input-specific block tracing would have to be set at around 3%. This might appear quite low, but the reader should bear in mind that these weighted percentages are intentionally skewed via a D^D weighting factor. Thus, a more plausible means of maintaining a threshold would be not to look solely at the weighted percentage that a procedure call attains, but to do thresholding based on a logorithmic trend among the weighted percentages. For example, viewing the weighted percentages collectively, those procedures whose weighted percentages are within an order of magnitude of the predominant percentage would be likely candidates for input-specific block tracing. With this strategy, all procedures in our pedigree trace at depths 8 and 9 would pass the threshold. This is what we probably want but it does not rely on an *ad hoc* fixed threshold like 3%.

Tracing

Even if none of the input data remain stable for a given bottleneck procedure, it may nonetheless be the case that the caller of this procedure remains consistent throughout many of the calls. By expanding the profiling to record more call graph trace-style information, this too could be exploited. For instance, attempting a specialization of a procedure for the case where it is frequently called by a specific caller (like, in our pedigree example, calling `P` almost exclusively inside of calls to `sc-*`) could result in improved register allocation for the callee. This alone might substantially improve performance. Witness the many efforts directed solely along these lines in the compiler optimization literature. For example, ACM PLDI has been rife with papers on register allocation since its inception to present.

At any rate, the specific mechanism whereby this could be achieved is fairly readily available in the MIT SCHEME compiler. Specifically, the RTL control flow graph (before RTL optimization) embodies a traditional graphical representation of the control flow. Profiling information could be collected and fed back into the graph as branch-frequency decorations. These, in turn, could be exploited by the RTL optimization phase to generate better register allocation decisions. Such recompilation techniques are becoming quite popular as they have proved to be effective at code performance enhancement. By using this technique in an adaptive multiple-specialization setting, I expect still better performance. Implementing this should involve a fairly modest and standard extension to the existing register allocation algorithm in the compiler back end and only a very slight augmentation of the control flow graph data structure (to carry the branch frequency information).[5]

A sample trace of the first 40 branches of our pedigree example, as generated by a naive prototype profiler, is shown in Figure 0-9. This reveals the actual path through the control/dataflow graph taken by the PC counter during a sample execution of the code. Specifically, it shows, for each procedure invoked, a pair of the name of the procedure and how much time was spent in the procedure body before transferring control to another code block. In a more realistic trace profiler (that is, one that is not a naive prototype), rather than maintain a never-ending trace of calls, a replica of the control flow graph would be maintained instead, which would be decorated to annotate how many times or with what frequency particular edges in the flow graph are taken by the executing program.

The control-flow based profiling strategy described in [Ball & Larus 92] was designed to promote branch frequency discovery in such a way that the optimized and otherwise convoluted object code actually executed will nonetheless support revelation of branch frequencies in the original, unmolested control flow graph. This is not altogether unlike the source-code/object-code de-sensitization strategy presented in [Appel, Duba & MacQueen 88].

Specifically, the `vector-summation` subroutine of our pedigree code produced the control flow graph in Figure 0-11 when compiled by the MIT SCHEME LIAR compiler (Release 7.2.0 α). It was (manually) graphically regenerated from the RTL code listing of Figures 0-12–0-21 (which were modestly massaged and cross-referenced for readability). For your convenience, the source code for `vector-summation` is provided again in Figure 0-10. The remainder of this subsection will provide a guided tour of these reams of code, focusing primarily on the details of `vector-summation`.

The `vector-summation` source code is a straightforward linear iteration of all indices of a vector, accumulating the sum of `proc` called on each element. Vectors in SCHEME are 0-indexed.

Before diving headlong into the bowels of the RTL code, let's take a peek at the control flow graph that results from compiling this procedure. It is shown in Figure 0-11.

The RTL (Register Transfer Language) code generated from this SCHEME source is developed through the next several pages. The LIAR SCHEME compiler assumes a registered stack machine architecture, meaning that arguments to and results from procedure calls are

[5] In fact, Andy Berlin has told me that he has a profiler fairly close to this for LIAR. He has not, however, attempted to feed this information back into the compiler: he uses it solely for programmer consumption.

```
((=======-TRACE-PROFILING-==========)
 (---p----4.1 . 0.)
 (---p----4.2 . 9.999999999990905e-3)
 (---p----7.1 . 0.)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 0.)
 (---sc-*-9.0 . 0.)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 0.)
 (---sc-*-9.0 . 9.999999999990905e-3)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 9.999999999990905e-3)
 (---sc-*-9.0 . 9.999999999990905e-3)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 0.)
 (---sc-*-9.0 . 0.)
 (v-sigma-8.0 . .01999999999998181)
 (---sc-*-7.0 . .01999999999998181)
 (---p----7.1 . 0.)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 0.)
 (---sc-*-9.0 . 9.999999999990905e-3)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 (---p----9.4 . 0.)
 (---sc-*-9.0 . .01999999999998181)
 (---p----9.1 . 0.)
 (---p----9.2 . 0.)
 (---p----9.3 . 0.)
 ...)
```

Figure 0-9: Pedigree example trace.

```
(define (vector-summation v proc)
  ;;-------------------------------------------------------------------------
  "(VECTOR-SUMMATION v proc) sums the invocation of PROC on each vector V elt."
  ;;-------------------------------------------------------------------------
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc            0            (+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))
```

Figure 0-10: Source code for `vector-summation` (from Figure 0-5).

passed on the stack. Specifically, the stack pointer is maintained in register `#x16`. It pushes towards higher addresses and always pointing directly at the last datum pushed.

Looking specifically at Figure 0-12 now, the most salient feature of this RTL listing is the small section marked `[**UNNECESSARY**]` (lines 11–15). It reflects a just paranoia on the part of the compiler that our vector indices may be larger than SCHEME can accommodate in a small fixed size number cell (a fixnum), so it always looks over our shoulder in case a bignum shows up. Of course, we know that it will not, but the compiler cannot infer that until it has witnessed the code in action, as we shall see.

One other interesting note is the region delimited by `BEGIN/END-DECR-INDEX` (lines 4–16). This highlights the code used to safely decrement the initial measured `vector-length` of `v`. Referring to the SCHEME source code listing for `vector-summation` (Figure 0-10), there are two textual occurrences of the SCHEME primitive decrement operator (`-1+`). Each occurrence will result in a similar block of RTL code, as will be seen shortly.

Proceeding to the entry into the `do` loop (Figure 0-13), we notice again a paranoid safety test by the compiler to again properly handle bignums in testing the `negative?` termination condition of the loop. Again, for the data we shall present, this paranoia is unjustified.

Next we see (Figure 0-14) the major fault of our `vector-summation` subroutine: it utilizes higher-order procedures. Specifically, an unknown `proc` is passed as a parameter to be used as an operator in the loop iteration. This can entail substantial performance penalties, not the least of which include the stack traffic involved in a general procedure call. Deciding on-the-fly to generate specialized versions of this code block for the most common (and most performance critical) choices of `proc` is the hallmark of this dynamically adaptive approach.

Specifically, specializing just the innermost call to `vector-summation` for its fixed argument `(lambda (g_m)...)` should prove very worthwhile, as Szolovits points out. Our profile-guided focus would lead us to the same conclusion, though perhaps not so emphatically: it would call for specialization of the inner multiply foremost, but as this is the body of the argument to `vector-summation`, our depth-based focus weighting strategy would draw in this most desired specialization in due course.

Next (Figure 0-15), we see a specific code block for an out-of-line generic arithmetic operator, in this case `+`. This corresponds to the computation of `(+ acc (proc (vector-ref v index)))` in the SCHEME source code. Here, (well, actually lines 25–28 of Figure0-14, which leads us to branch to Figure 0-15) because we are dealing with conditional probabilities (which are

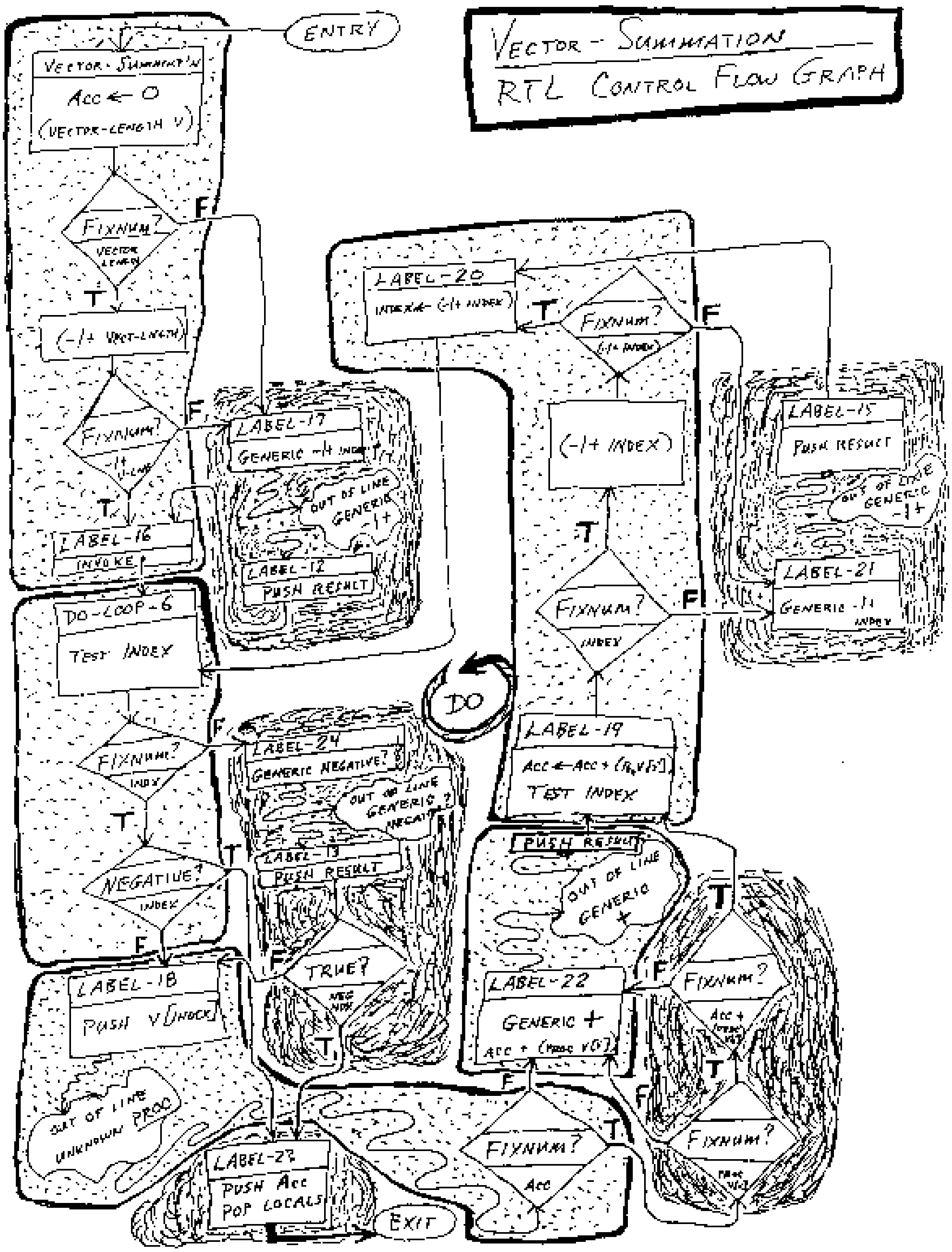


Figure 0-11: Vector-sum control flow graph.

```
;;;-------------------------------------------------------------------------------------------
;;;                         RTL for VECTOR-SUMMATION
;;;-------------------------------------------------------------------------------------------
((procedure-header vector-summation-10 3 3)
;;; Temp constant just for kicks
 (assign (register #x30) (constant 0))
;;; push local acc <- 0
 (assign (pre-increment (register #x16) -1) (register #x30))
;;; push local index <- (-1+ (vector-length V))                    ;==BEGIN=DECR-INDEX=======v
 (assign (register #x32) (machine-constant #x1A))
 (assign (register #x33) (offset (register #x16) 1))                ; V
 (assign (register #x34) (object->address (register #x33)))
 (assign (register #x35) (offset (register #x34) 0))                ; V's length
 (assign (register #x31)
         (cons-pointer (register #x32) (register #x35)))
(jumpc (not (type-test (register #x32) #x1A)) label-17)            ; Figure 0-19
;;;-------------------------------------------------------------------------------------------
;;; potential out-of-line call to generic -1+ of (vector-length V)   ; [**UNNECESSARY**]
(assign (register #x3C) (object->fixnum (register #x31)))
(assign (register #x3B)
        (fixnum-1-arg minus-one-plus-fixnum
                      (register #x3C) #t))
(jumpc (overflow-test) label-17)                                   ; Figure 0-19
;;;-------------------------------------------------------------------------------------------
;;; push local index
 (assign (register #x3D) (fixnum->object (register #x3B)))        ;==END===DECR-INDEX=======^
(assign (pre-increment (register #x16) -1) (register #x3D))
(jump label-16)                                                    ; Figure 0-13
;;;-------------------------------------------------------------------------------------------
```

Figure 0-12: RTL Vector-Summation Entry Code

```
;;;-----------------------------------------------------------------------------------------
;;; do loop initial entry
(label label-16)
(invocation:jump 2 ()  do-loop-6)  ;--.
;;;-----------------------------------------------------------------------------------------
(open-procedure-header do-loop-6)  ;<-'
;;; out of line generic NEGATIVE? of index                    ; [**UNNECESSARY**]
(assign (register #x30) (offset (register #x16) 0))
(assign (register #x31) (object->type (register #x30)))
(jumpc (not (type-test (register #x31) #x1A)) label-24)        ; Figure 0-21
;;;-----------------------------------------------------------------------------------------
;;; exit do loop... return acc
(assign (register #x4A) (object->fixnum (register #x30)))
(jumpc (fixnum-pred-1-arg negative-fixnum?
                          (register #x4A)) label-23)           ; Figure 0-20
(jump label-18)                                                ; Figure 0-14
;;;-----------------------------------------------------------------------------------------
```

Figure 0-13: RTL Vector-Summation Exit Test Code

traditionally represented as floating-point numbers in the range $[0, 1]$), we will nearly always have to resort to out-of-line generic floating-point (flonum) addition at this point. Thus, unlike the previous `[**UNNECESSARY**]` fix/big/flonum paranoia, this flonum precaution (namely, line 28 of Figure 0-14) is `[**VITAL**]`. Nevertheless, notice that the vain attempt to cash in on faster fixnum operators (in lines 26 and 27 of Figure 0-14) will be repeated with every iteration of the loop. This is a fool's errand. Again, by dynamically specializing this routine for a given pedigree tree, this futile hedging could be avoided. Although this really means eliminating only these two lines and making the conditional jump (line 28 of Figure 0-14) become an unconditional jump, avoiding even these three lines of RTL code in the innermost loop could make a measurable difference when the loop is invoked a large number of times for a very large pedigree.

These last two figures were the most important for dramatizing the benefits of the approach I propose. It is these two code blocks, at the heart and center of the do-loop iteration, which are the most expensive. Together they embody two of the most renowned architectural horrors that FORTRAN coders like to foist upon LISP programmers: 1) first-class procedures entail out-of-line procedure calls with dynamic linking (usually stack-based) and 2) the nifty generic arithmetic of LISP-languages can incur marked overhead due to unnecessary or obvious type checks followed by conditional branches. By assuaging these objections, the performance of our LISP-like code should asymptotically approach that of FORTRAN and Clike code. Andy Berlin [Berlin 90] accomplished essential this for numeric LISP-like code. I intend comparable success with generic and higher-order procedures.

These blocks also demonstrated the major specific motivations for the adaptation techniques I develop in the next section. First, finding heavily trafficked paths through the code will be used to specialize sequences of blocks or procedures which are performance

```
;;;-------------------------------------------------------------------------------------------
(label label-18)
;;; do loop continue... (proc (vector-ref v index))
(assign (register #x39)                                          ; push do loop header
        (cons-pointer (machine-constant #x28)
                      (entry:continuation continuation-3)))      ; [below]
(assign (pre-increment (register #x16) -1) (register #x39))
;;; compute acc + (proc (vector-ref V index))
(assign (register #x3B) (offset (register #x16) 3))              ; V
(assign (register #x3D)                                          ; V start address
        (address->fixnum (object->address (register #x3B))))
(assign (register #x40) (offset (register #x16) 1))              ; index i
(assign (register #x42)                                          ; offset: header to i
        (fixnum-2-args multiply-fixnum
                       (object->fixnum (constant 4))
                       (object->fixnum (register #x40)) ()))
(assign (register #x43)                                          ; V[i] addr arith
        (fixnum-2-args plus-fixnum
                       (register #x3D)
                       (register #x42) ()))
(assign (register #x3A) (fixnum->address (register #x43)))       ; addr of V[i]
(assign (register #x45) (offset (register #x3A) 1))              ; read V[i]
(assign (pre-increment (register #x16) -1) (register #x45))      ; push V[i]
(assign (register #x46) (offset (register #x16) 5))              ; read proc
(assign (pre-increment (register #x16) -1) (register #x46))      ; push proc
(invocation:apply 2 continuation-3)                              ; call (proc V[i])
;;;-------------------------------------------------------------------------------------------
;;;     ------- Big Hairy Expensive Out of Line Procedure Call Here -------
;;;-------------------------------------------------------------------------------------------
(continuation-header continuation-3)                             ; rtn from proc call
(assign (register #x30) (register 2))                            ; pop result of call...
(assign (register #x37) (offset (register #x16) 1))              ; read acc
(assign (register #x38) (object->type (register #x37)))          ; acc + (proc V[i])...
(jumpc (not (type-test (register #x38) #x1A)) label-22)          ; Figure 0-15 [**VITAL**]
(jump label-rare-win)                                            ; Figure 0-16
;;;-------------------------------------------------------------------------------------------
```

Figure 0-14: RTL Vector Summation acc Update

```
;;;-----------------------------------------------------------------------------------------
;;; out of line generic + of
;;; acc [flonum] and (proc (vector-ref V index)) [flonum]          ; [**VITAL**]
(label label-22)
(assign (register #x31) (offset (register #x16) 1))
(assign (register #x35)
        (cons-pointer (machine-constant #x28)
                      (entry:continuation label-14)))              ; [below]
(assign (pre-increment (register #x16) -1) (register #x35))
(assign (pre-increment (register #x16) -1) (register #x30))
(assign (pre-increment (register #x16) -1) (register #x31))
(invocation:special-primitive 3 label-14                           ; [below]
                              #[primitive-procedure &+])
;;;-----------------------------------------------------------------------------------------
;;;           ----- Big Slow General Generic Procedure Call Here -----
;;;-----------------------------------------------------------------------------------------
(continuation-entry label-14)
(assign (register #x36) (register 2))
(jump label-19)                                                    ; Figure 0-18
;;;-----------------------------------------------------------------------------------------
```

Figure 0-15: RTL Vector Summation Out-of-Line Generic ADD

bottlenecks. Second, data-specific optimizations (especially fix/big/flonum specializations) will be pursued. Finally, specialization with respect to specific constants will be performed, such as the constant fixnum range for the index in the pedigree example.

One last technique is not really applicable with my trivial definition of $P(\ldots)$ but would become an issue were we to implement it as a lookup into a table of conditional probabilities (as Szolovits correctly does). That technique is to on-the-fly manage memoized versions of key side-effect free code blocks. In Szolovits implementation, memoization of the calls to $P(\ldots)$ is an obvious first step.[6]

Thus, through profile-based focus, I propose to alleviate the above-mentioned central haunting objections to LISP-like languages through judicious aggressive optimization, essentially by making on-the-fly specializations of code blocks to make them correspond to the FORTRAN code one might have written. That is, I make it the burden of the compiler and run-time system, *not the burden of the programmer*, to optimize the very generic LISP object code into something more. Thus, one should be able to simultaneously attain the expressibility and flexibility that is desired with the performance that is needed *where it matters most*.

Anyway, for completeness, Figure 0-16 shows the much desired but seldom attained performance improvement in the rare circumstance where the `acc` and `(proc V[i])` both

[6]In fact, memoization was assumed by Szolovits in his presentation of this example. Unlike Szolovits, however, I do not anticipate generating the entire conditional probability matrix then using table dispatch to access these values. Instead, I would leave the table implicit in the adaptively managed memoized accessor procedure `P`.

```
;;;-------------------------------------------------------------------------------------
(label label-rare-win)
(assign (register #x39) (object->type (register 2)))
(jumpc (not (type-test (register #x39) #x1A)) label-22)        ; Figure 0-15 out of line +
;;;-------------------------------------------------------------------------------------
(assign (register #x3C) (object->fixnum (register #x37)))
(assign (register #x3D) (object->fixnum (register 2)))
(assign (register #x3A)
        (fixnum-2-args plus-fixnum
                       (register #x3C)
                       (register #x3D) #t))
(jumpc (overflow-test) label-22)                                ; Figure 0-15 out of line +
(jump label-assign-acc)                                         ; Figure 0-17
;;;-------------------------------------------------------------------------------------
```

Figure 0-16: RTL Vector Summation `acc` Rare Win

yield fixnums. In fact, this should occur only on the initial call to the loop, when `acc` is 0 and the `proc` code has some prayer of returning a fixnum (say, reflecting a certain probability of an empirically observed ("known") parameter). Indeed, balancing the cost of the optimistic test for fixnums against the branch frequency with which it is successful is the grounds for deciding, dynamically, to spawn a specialized version of this code block for use in the pedigree setting. Specifically, if branch frequency indicates that the fixnum jump is taken only 1% of the time and that the difference in execution speed of the fixnum case from the non-fixnum case is slight, then the cost of executing the type dispatch (the type test and subsequent conditional jump) in the remaining 99% of the time may well offset the total time gained by all successful fixnum jumps. In such a situation, the LIAR compiler's generated type specialization on fixnums may be deemed uneconomical, so the code may be specialized to just always call the generic `+` instead. Note, by the way, that the LIAR compiler can of course statically determine what this threshold frequency will be so the run-time system need only check this statically deduced threshold (probably something like 90% fixnum miss makes the fixnum testing uneconomical) against the dynamically observed branch frequency. This is an archetypal instance of a situation where the compiler might generate aggressive code but install reality checks for the run-time system to dynamically verify, if appropriate, to try to fine-tune the code performance in critical bottleneck regions.

```
;;;-------------------------------------------------------------------------------------
(label label-assign-acc)
(assign (register #x36) (fixnum->object (register #x3A)))     ; tmp-36 <- acc + (proc Vi)
(jump label-19)                                                ; Figure 0-18
;;;-------------------------------------------------------------------------------------
```

Figure 0-17: RTL Vector Summation `acc` Assignment

Also for completeness, Figure 0-17 shows the actual assignment block for the iterant value of `acc`. Were the code to be specialized for in-line floating-point add here, this block would naturally be absorbed into that specialization since there would no longer be more than one control flow entry edge into this block.

```
;;;-----------------------------------------------------------------------------------------------
;;; store acc <- acc + (vector-ref V index) ; compute (-1+ index)
1 (label label-19)
2 (assign (offset (register #x16) 1) (register #x36))           ; acc <- <tmp-36>
3 (assign (register #x43) (offset (register #x16) 0))          ;==BEGIN=DECR-INDEX=======v
4 (assign (register #x44) (object->type (register #x43)))
5 (jumpc (not (type-test (register #x44) #x1A)) label-21)       ; Figure 0-21 out of line -1+
;;;-----------------------------------------------------------------------------------------------
6 (assign (register #x47) (object->fixnum (register #x43)))     ; [**UNNECESSARY**]
7 (assign (register #x45) (fixnum-1-arg minus-one-plus-fixnum
8                                       (register #x47) #t))
9 (jumpc (overflow-test) label-21)                              ; Figure 0-21 out of line -1+
;;;-----------------------------------------------------------------------------------------------
10 (assign (register #x42) (fixnum->object (register #x45)))    ; tmp-42 <- (-1+ index)
;;;-----------------------------------------------------------------------------------------------
11 (label label-20)                                             ; index <- <tmp-42>
12 (assign (offset (register #x16) 0) (register #x42))         ;==END===DECR-INDEX=======^
13 (invocation:jump 2 () do-loop-6)                             ; Figure 0-13 continue
;;;-----------------------------------------------------------------------------------------------
```

Figure 0-18: RTL Vector Summation `index` Update

Next (Figure 0-18) we see the index iteration update code. As with the loop entry (back in Figure 0-12, lines 4–16), observe the unnecessary paranoia (Figure 0-18, lines 5 and 9) concerning the index overflowing into a bignum or even being a flonum. Notice, moreover, that this is an entirely redundant test since the fixnum-itude of the index is tested in the termination test at the start of each loop iteration. Although such a picky flow-analysis optimization would unlikely prove critical to a lesser exercised code block, the simple improvement of registering the results of those earlier type tests could enhance performance in such a bottleneck as the innermost occurrence of `vector-summation` in our sample pedigree computation. Such aggressive flow analysis is seldom worth while in a general setting, but upon detailed monitoring and analysis, it may be deemed justifiable in a dynamically adaptive setting. Specifically, once the dynamic flow graph has been ascertained through profiling and some branch frequencies have been computed, walking along the heavily traveled paths of the graph (especially the loop bodies) and eliminating redundant or semantically subsumed type and value tests may prove beneficial. Such optimizations are very seldom apparent statically.

At any rate, for completeness, Figure 0-19 shows the generic out-of-line `+` that would be invoked should our index ever overflow.

And lastly, Figure 0-20 shows the do-loop exit code, which simply pushes the result (`acc`) onto the stack and returns to the caller whose return address is likewise on the stack.

Finally, just to tie up any loose ends, the scaffolding of a couple other out-of-line generic procedure blocks are presented in Figure 0-21. No more insight would be gleaned from

```
;;;-------------------------------------------------------------------------------------------
;;; out of line generic -1+ for (vector-length V)                  ; [**UNNECESSARY**]
(label label-17)
(assign (register #x39)
        (cons-pointer (machine-constant #x28)
                      (entry:continuation label-12)))              ; [below]
(assign (pre-increment (register #x16) -1) (register #x39))
(assign (pre-increment (register #x16) -1) (register #x31))
(invocation:special-primitive 2 label-12                           ; [below]
                              #[primitive-procedure -1+])
;;;-------------------------------------------------------------------------------------------
 (continuation-entry label-12)
 (assign (pre-increment (register #x16) -1) (register 2))
 (jump label-16)                                                    ; Figure 0-13
;;;-------------------------------------------------------------------------------------------
```

Figure 0-19: RTL Vector Summation Out-of-Line Generic DECR

```
;;;-------------------------------------------------------------------------------------------
;;; exit do loop... return acc on the stack
(label label-23)
(assign (register 2) (offset (register #x16) 1))                   ; push acc
(assign (register #x16) (offset-address (register #x16) 4))        ; pop locals
(pop-return)
;;;-------------------------------------------------------------------------------------------
```

Figure 0-20: RTL Vector Summation Return Code

```
;;;-------------------------------------------------------------------------------------------
 (label label-21)
 ;; Not shown... out of line generic -1+ of index                  ; [**UNNECESSARY**]
 ;;   this code block is unsettlingly similar to that of label-17  ; Figure 0-19
 (jump label-20)                                                   ; Figure 0-18
;;;-------------------------------------------------------------------------------------------

;;;-------------------------------------------------------------------------------------------
 (label label-24)
 ;; Not shown... out of line generic NEGATIVE? of index            ; [**UNNECESSARY**]
 ;;   there is really nothing new or special to see here.
 (jump label-23))                                                  ; Figure 0-20
;;;-------------------------------------------------------------------------------------------
```

Figure 0-21: RTL Vector Summation Fragments showing the gory details of these fairly typical blocks, so I just show where they will virtually always return to, if in fact called.

This finally completes the full RTL control flow graph. (For convenience, the RTL control flow graph is again presented in Figure 0-22.)

The RTL listing and the control flow graphs for `sc-*` are annoyingly similar to those of `vector-summation`. (Witness again the source code listing for `vector-summation` and `sc-*` in Figure 0-23.) The interesting difference, however, is that the source of most of our concerns in `vector-summation` stemmed from the first-class procedure parameter `proc`. Well, here the problem is further exacerbated since we introduce a further level of indirection: rather than receive just a procedural parameter, `sc-*` receives a data structure containing procedures to be invoked. Again, however, profiling should reveal a persistent stability in this data structure (it is never mutated) so again on-the-fly specialization may indeed be in order. In fact, the adaptive system may well learn to recognize certain patterns of adaptation which almost always prove worthwhile so, given time, the system may learn to always instantly spawn specialized blocks for this procedure. This may well serve as the telltale sign for the need of a more structural algorithmic transformation of the source code. Reporting such a discovery to the interested code developer/maintainer should prove a valuable service.

The presentation now returns to a more general and far reaching scope than this detailed scenario section. It also elaborates on some of the open concerns and potential difficulties in the proposed approach. Most important among these is the issue of generating and maintaining multiple specializations of a procedure.

0.7.4 Dynamically Adaptive Specialization: The Specifics

Imagine now that we have conducted full profiling on a piece of code so that we now know what procedures are the bottlenecks. Further, imagine we have even done input-specific block tracing, so for each such bottleneck we will have histogram information about what the input arguments were upon each invocation. In the case where some commonality exists in the frequently encountered argument lists, we would like to generate specialized versions of the procedure being called.

For instance, if most of the time the second argument to some multi-argument procedure is 42, we might like to generate a specialized version of this procedure for that special case. Thereafter, whenever we call the procedure, we would first test the second argument to see if it is 42. If it is, we would immediately jump to the specialized procedure body; otherwise we may proceed with the unspecialized (original) procedure body. In the general case of multiple specializations, we might proceed with an N-way dispatch to the appropriate specialized body based on some distinguishing characteristics of the argument list as a whole. Less trivial argument specializations might include type specializations on some arguments with simultaneous value specializations on others. For instance, some procedure might well be specialized for the situation where its first argument is 0 (some common initial value), its second argument is a power of 2 (some nice binary block size parameter), and its third argument is null (an optional argument not supplied).

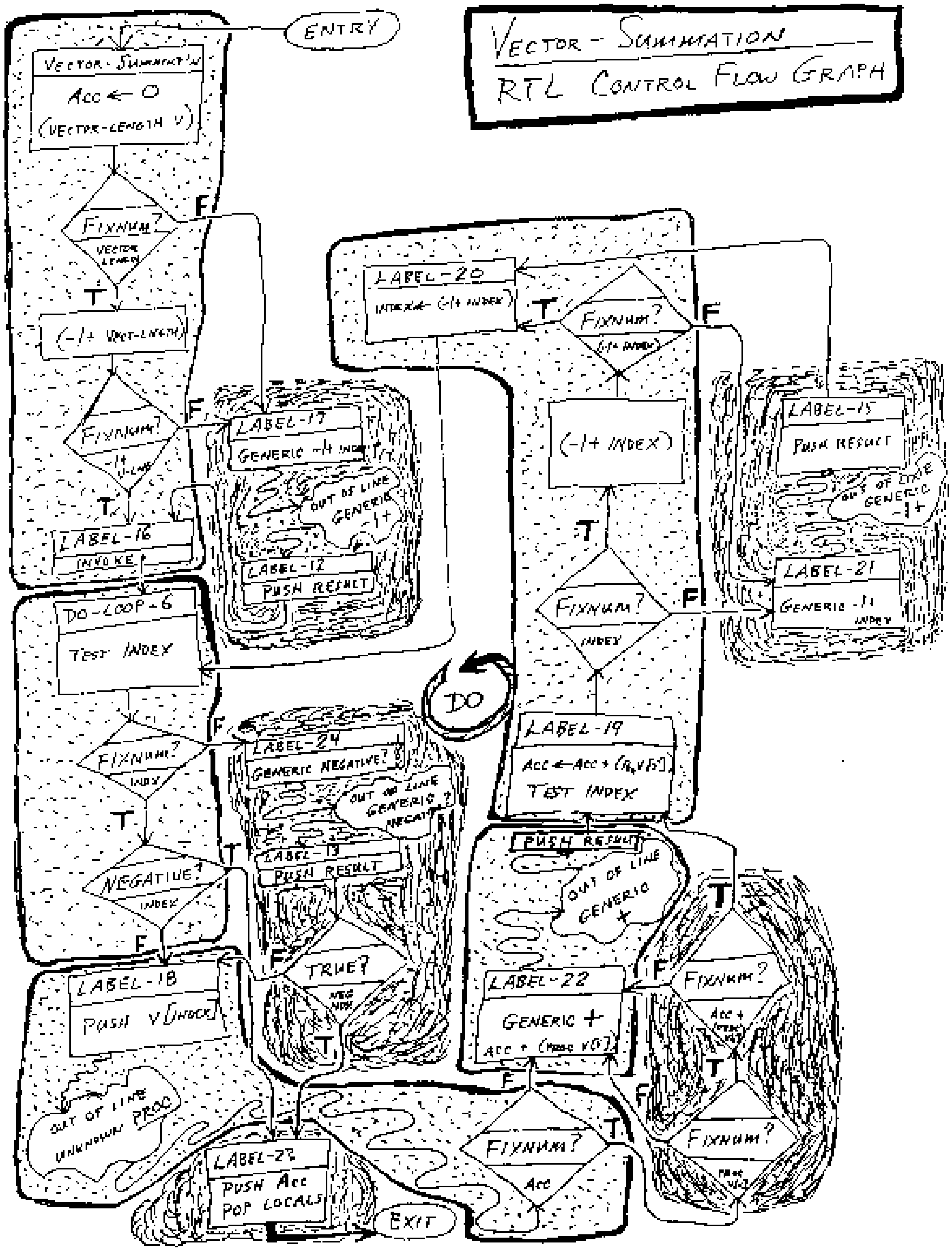


Figure 0-22: Vector-sum control flow graph— again.

```
;;;
;;; Pedigree Utility Subroutines --- again.
;;;

(define (vector-summation v proc)
  ;;-----------------------------------------------------------------------------
  "(VECTOR-SUMMATION v proc) sums the invocation of PROC on each vector V elt."
  ;;-----------------------------------------------------------------------------
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc             0             (+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))


(define (sc-* . thunks)
  ;;-----------------------------------------------------------------------------
  (string-append
   "(SC-* . thunks) short-circuit times of thunk calls from left-to-right."    newline-string
   "If any thunk call yields 0, short-circuits to return 0 w/o calling thunks" newline-string
   "to its right.")
  ;;-----------------------------------------------------------------------------
  (do ((thunks-2-do thunks (cdr thunks-2-do))
       (acc           1    (* acc ((car thunks-2-do)))))
      ((or (null? thunks-2-do)
           (zero? acc))
       acc)))


(define (P val . givens)
  ;;----------------------------------------------------------------------------------
  "(P val . givens) computes the conditional probability of VAL given all GIVENS values."
  ;;----------------------------------------------------------------------------------
  ;; This would actually do a table lookup in a table of empirically observed
  ;;  conditional probabilities.
  ;; To keep things simple for now, we just effectively return VAL.
  (let ((given-val (reduce * 1 givens)))
    (/ (* val given-val)
       given-val)))
```

Figure 0-23: Pedigree computation utility subroutines.

More on Problem 2: Control Stability (page 15)

The open problem, of course, is deciding how to take a procedure and a histogram of dynamically observed argument lists for that procedure then deduce or otherwise recognize some useful common structure among the arguments that could be exploited to enhance the procedure's performance in practice. Some static type-based work has been done in this area at Stanford by the FUSE project by Ruf and Weise [Ruf & Weise 91] in the setting of avoiding redundant specializations and additional work has been done by the SELF group (See papers by Ungar and Chambers) in the setting of specializing for object-oriented method dispatch. Both these efforts have been based strictly on type information. I propose to generalize this to value-based specialization and property-based specialization. In that regard, the literature on learning from examples [Mitchell 81], or explanation-based learning, may provide a more general framework to explore the specifics of learning dynamic procedure usable patterns from the examples that their input arguments provide. This whole general extension of the work of FUSE and SELF might be left as a future direction that this work could pursue. It is probably the sort of highly aggressive dynamic optimization that would best be relegated to after-hours off-line code massaging anyway.

Aside from deciding precisely what specializations might be pursued, actually choosing from among these alternatives precisely which one to execute and how many of them (if any) to dynamically realize is a separate concern. Specifically, the decision of whether or not to perform these specializations must balance the cost of performing the specialization against the expected benefit of having done so. This can be empirically decided by the run-time optimizer by first deciding that it should heuristically be worthwhile to at least investigate the possibility. For instance, a heuristic threshold built into the optimizer would trigger suggesting that some particular candidate for multiple specialization should at least be granted one specialization for one of the apparently stable input configurations. For example, if four different specializations might be pursued with respect to just the first argument being a known value (*e.g.*, , it may range over the interval $[0, 3]$), then at least one specialization (say, for 0) could be realized. Subsequent tracing would reveal how much was gained by this specialization. If the benefit (measured speedup) outweighs the cost of performing the optimization, then this would supply supporting evidence that the remaining prospective specializations might be fruitfully undertaken. Some of this decision can be made based on static information, such as using conservative estimates on the clock cycle of each instruction to anticipate a rough comparison between the code currently executing and the code that the specialization would produce. (Of course, it may be cheaper just to do the specialization and measure the performance difference rather than perform some abstract interpretation to compute estimated performance.) This static information can be used to provide *a priori* supporting evidence for or against attempting a specialization. Afterward, the *a posteriori* empirical evidence measured for the true performance of a prospective additional specialization provides the supporting evidence whether this investment strategy (that is, set of related prospective specializations) should be continued.

The strategy above attempts to hesitantly pursue specialization without over-specializing. It first tries one promising looking specialization then tries others as evidence mounts in their favor. If an initial specialization proves disappointing, however, one way of gracefully backing out of the aggressive specialization is to try to coalesce (generalize) the set of proposed specializations. For instance, if a few arguments are always stable while a few others fluctuate a little while still others fluctuate quite a bit, deciding how many specializations to attempt must be balanced against whether the resulting specializations will prove fruitful over time. Considering the selection of which specializations to pursue and which not to as being fundamentally a partitioning problem, such a situation would chose some "reasonable" initial partition and if it proves unfruitful, the many prospective specializations could be coalesced into fewer, less aggressive, more general prospective specializations which should have a better chance of being cost effective (since they would be applicable in a larger set of observed argument instances). In the worst case, this could all degenerate to the case of taking all invocations as a whole and attempting specialization of this aggregate group.

Return to Problem 1: Data Stability (page 14)

Beyond the somewhat static (instantaneous) issue of deciding what alternatives exist for multiple specialization and which, if any, of those alternatives should be pursued, there remains too the dynamic concern of how to maintain these collections of multiple specializations in the face of an ever-changing run-time environment. That is, how should these collections of specializations, viewed as a population of optimized procedures, adapt to the ever-changing run-time climate?

Returning again to the pedigree example (and to our heretofore neglected Problem 1 of page 14, consider now that we have optimized the nested loops as much as appears justified (and have consequently turned off the aggressive profiling on those procedures which we have already optimized to our satisfaction). We turn our attention now to the issue of conducting several experiments with the same basic pedigree tree, perhaps by mutating some of the "known" nodes or by even introducing newly obtained subtrees.

The above discussed specialization strategies all focused on exploiting stabilities in the input data based on the data's values. By contrast, we now consider the more general case of input data which remains structurally stable (its shape doesn't change) but whose immutable component values vary widely. A trivial example would be a 5-element list of integers that remains a 5-element list of integers but whose component integers change over the course of a computation.

Now, imagine that, in light of the preliminary results produced from the pedigree tree first presented in Figure 0-1, we are prompted to collect additional information for our tree. Specifically, imagine that we have the matriarch of the pedigree tree available for a genotype test (a T_F test node). Assume further that the daughter (the D subscripted nodes) has married and has born a son. These new nodes provide improved information on which to compute the probabilistic pedigree expression, but the most important point is that *none of the old pedigree nodes have changed.* Thus, although much of the underlying equation will be affected by the presence of the new nodes (since the conditional probabilities of other nodes' expression must now take into consideration the new constraints imposed by the newly available evidence), much of the old equation will nevertheless remain the same. Thus, much of the optimizations adapted for the old pedigree tree should remain valid and beneficial.

Finding a way for newly presented data to be recognized as structurally similar to formerly witnessed data is an open area of research. Nonetheless, the looping structure of the new equation will be substantially similar to the old equation, including the loop nesting structure and the stability properties of the procedure calls. Thus, recording the decision criterion used in establishing the present specializations could prove valuable in adapting to the new input.

This suggests some concerns of dynamically expanding aggregate data (like trees that grow). Perhaps more challenging is the problem of retracting data (trees that shrink). Specifically, imagine that in the initial tree, the nodes associated with the daughter (the D subscripted nodes) were mere conjecture. Imagine now we wish to retract them altogether. This will naturally result in the removal of several subterms of the original equation. How this will impact the collection of specializations generated is tricky. At least in the case of expanded data, most sub-components of the original data were still present so their specializations could be treated as (say) curried specializations with respect to the new data. The removal of data is not as easy to deal with.

Finally, conducting new experiments which simultaneously retract some nodes while expanding others compounds the difficulties of both situations individually. Depending on how retractions alone are treated, it may not even be equivalent to first expand then retract versus first retracting then expanding.

0.7.5 Final Comments on the Pedigree Example

In closing on this example, one should be carful to note that it is the steady-state performance of the code which I aim to optimize, so I can use a somewhat loose optimization investment strategy since its costs will amortize over the lifetime of the optimized procedure. So long as the system successfully substantially improves the performance of the critical bottlenecks, while avoiding tweaking the remaining code, then it should easily pay for its overhead, even if it fails to do an "optimally optimal" job of optimizing.

0.7.6 Problems and Open Issues: The Specifics

Other issues not raised by the above example also will need to be addressed. The most prominent of these remaining concerns follow:

The Cost/Benefit Investment Strategy

I have not said anything specific about the cost/benefit assessment of various optimizations. For instance, deciding to profile a segment of code has an associated cost, as does actually performing the profiling once the decision to do it has been made. Some compile-time criteria for detecting when this overhead is likely to result in some cost-offsetting optimization should be used to direct the thresholding on increases profiling. Specifically, some sort of prioritization on the classes of optimizations that are applicable will have to be empirically determined. These must be based on a careful analysis of the efficacy of various traditional optimizations. The forthcoming dissertation by Craig Chambers (as yet unpublished but ftp-able from `self.stanford.edu`) will likely prove the core of this assessment.

Passing Procedures as Parameters

Optimizing one instance of a procedure as it occurs in one context does not affect other textual occurrences of the procedure in other blocks of code. On occasion, however, it may be desirable to perform an optimization globally on all instances of a procedure. This is not dissimilar to the problem of optimizing an in-line coded procedure, so this problem arises even in the non-higher-order setting. Some indirection mechanism may alleviate this concern by introducing controlled sharing among procedure instances.

Mutability and Memoization

A more subtle concern involves mutation of data with which a procedure or procedures may have been specialized. Structurally mutating an input (for example, using `set-cdr!` to dynamically add a node to a tree structure) can invalidate several specializations (such as memoizations based on the `cdr` of the mutated structure). Traditional partial evaluation strategies avoid this issue altogether by either restricting themselves to a purely functional language or by restricting the partial evaluation to a purely functional subdomain of the language. Having the benefit of dynamically available information, however, we can opt to generate specializations nonetheless, then use the dynamic information of whether a datum has actually mutated to either choose or not choose to execute subsequent specialized procedures. For instance, if one side of a conditional performs a specialization-disrupting mutation but is rarely taken (as revealed by branch frequency profiling), we may choose to generate the specialization nonetheless. In the rare instance where the disruptive mutation is actually performed, we may choose to invalidate the specialization, reverting to the original still-valid code. I am assuming here that the specialization dispatcher would test occurrences of aggregate inputs using SCHEME's `eq?` predicate. This is why mutations of the list cells would require invalidation of the specialization. Alternatively, the dispatcher could use a more mutation-insensitive equivalence predicate such as `equal?`, but this would likely prove prohibitively expensive. Of course for value properties (like `(odd? (car arg1))`) would be insensitive to mutation in the sense that they would never falsely trigger as the result of a mutation.

One can think of this as maintaining within each procedure entry a dispatch table of all the various specializations of the procedure, where the dispatch is performed based on testing the input data. Generating a new specialization merely adds a test case to this dispatch table. Invalidating a specialization is then just a matter of removing a specialization from the dispatch table. (Think of it as analogous to invalidating a software cache entry). Some time afterward (determined, for instance, during the initial specialization), we may choose to re-specialize the same procedure based now on the mutated yet apparently temporarily stable datum.

As mentioned earlier, keeping a record of what decision led to which specializations might be very useful. In this situation, the basis of the selection of specialization alternatives can be codified to capture, for instance, what potentially mutable inputs were involved in the specialization and what properties of these involved mutable data were exploited. For instance, recording that the shape of the pedigree tree is used to generate the specialization of the very general equation for pedigree expression, any shape change to the input pedigree tree can invalidate any memoizations performed with respect to the tree. The problem here, of course, stems from the fact that doing `set-cdr!` to a pair alters the object contained in the cdr cell of the pair but it does not alter the pair of cells that the pair pointer points to (that is, it does not alter the address of the cell). It is this pointer (address) that `eq?` tests.

Clearly, this issue deserves more careful thought. For the time being, however, we can either consider that the early stages of this work will be restricted to non-mutable aggregate list structures (that is, we will only specialize with respect to an aggregate if the compiler/run-time system can prove or otherwise detect that the datum is a constant). Later phases can address the aggregate mutability quagmire. If nothing else, we will certainly do no worse than traditional methods which make no effort to specialize or adapt whatsoever based on structure stability anyway.

Adaptation and Phasing

As discussed at the end of the scenario section, handling dynamic adaptation of code could prove very difficult. Aside from finding ways to recognize patterns of stability in data, there may be pressure to recognize patterns of change as well. Specifically, if it is deemed useful to maintain no more than 5 specializations of a given procedure but those 5 specializations keep adapting in a recognizable way, it may be beneficial to either blend the adaptation so that 5 specializations are maintained which skirt the commonality of the adaptation pattern or else some split into a larger number of specializations might be worthwhile. For example, a program might cycle over a structured data performing a repeated smoothing or structural massaging. This may involve repeated expansion and retraction of the structure.

For instance, symbolic algebra rule sets exhibit this sort of "phasing" behavior by doing a repeated generate-and-test "smoothing" (that is, algebraic simplification) of their intermediate results. For example, MACSYMA and its successors employed an elaborate strategy of "reduce, simplify, reduce, simplify, ...". This constant adjustment to the aggregate data, in order to damp the semantic complexity of the algebraic expression it represents, results in data that is constantly changing, although in predictable ways. That is, the same simplification rules are likely to be applied repeated to the same list structure, but the list structure will constantly change in shape.

If the phases are sufficiently temporally separated such that the data remains stable for a long time within any given phase, we should have little difficulty. Similarly, if the data remains stable for only a relatively short time we also have no real problem since the run-time system will, with the aid of the profiler, be able to detect the fluctuation and attempt to mediate it with less aggressive specializations to skirt the commonality.

Unfortunately, if the phases find some natural resonant frequency of the run-time adaptation algorithm, we may find ourselves chasing our tail by continually investing in short-lived specializations which never quite pay in the long run. Some damping or smoothing mechanism may be necessary. As an alternative, some resonance detection mechanism may be possible. With such a mechanism, if deemed fruitful two sets of specializations may be split off for the two distinct phases recognized. If not deemed fruitful, perhaps only one of the phases could remain specialized (the bottleneck-dominant one) while the other could stay with non-specialized procedures.

This naturally hinges on the fact that the profiler cannot allow itself to accumulate arbitrary amounts of profiling data. It must, instead, maintain only a limited and compressed set of information, perhaps expanding and contracting at times based on the aggressiveness with which it chooses to pursue a particular optimization. Thus, temporal oscillations in data or flow control poses a problem to any spatially limited profiling since in limited space one cannot accumulate enough information to recognize long-term patterns of behavior.

This may, in fact, not prove to be a realistic concern in practice since it is not easy to imagine code that would exhibit this sort of long-term oscillation. At least, any such program would likely bear an underlying generate-and-test control structure like PROLOG-ish call-by-unification or a MYCIN-like rule-based "expert system". It is unclear that anything can be done to help such systems since they exhibit a fundamentally oscillatory control structure which one might dub "inherently unstable". Any specialization mechanism must fundamentally be based on exploiting patterns of stability, so it is not surprising that they offer little relief in unstable systems. At the very least, we can try to gracefully bail out of unstable oscillations without wasting too much time and effort in the futile pursuit of stabilizing them.

0.8 Required Facilities and Resources

This research will be conducted in the Scheme dialect of Lisp. A high-performance workstation which can execute Scheme programs will therefore by required.

0.9 Expected Time Table

```
Fall   92: PhD Thesis Proposal completed
Winter 92: Implementation of profiling tools for LIAR (MIT Scheme)
Spring 93: Initial development of profile-driven specialization

Summer 93: Examination of multiple-specialization issues

Fall   93: Development of economic cost/benefit model
Winter 93: Assessment of the efficacy of this approach
Spring 94: Re-evaluation of the thesis and assessment of future work

Summer 94: Composition of the dissertation
```

[This page intentionally left very nearly blank.]

Appendix A

Pedigree Compilation by P. Szolovits (Reprint)

Reprint of *Compilation for fast calculation over pedigrees* [Szolovits 92]

[This page intentionally left very nearly blank.]

Compilation for Fast Calculation over Pedigrees

Peter Szolovits, Ph.D.[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, 02139, USA

Efficient computation of probabilistic relationships over family pedigrees is an important tool for a variety of problems in genetics, including genetic counseling and linkage analysis. The development of faster and more comprehensive algorithms has preoccupied geneticists for decades [3], and recent general-purpose algorithms for probabilistic inference over arbitrary networks (e.g., [6]) have also been proposed as useful tools for such problems [8].

We present a set of engineering speed-up methods developed as part of the implementation of a prototype program for assisting genetic counselors, GENINFER-II. The design of the underlying computational algorithm is due to Cooper [2], and treats the problem of finding an efficient way to evaluate a Bayes network as a problem of factoring algebraic formulæ. It is thus an extension of the method of [3] (though I believe it was independently formulated) and handles consanguinity by extending the factoring method rather than by cutset conditioning (as does, for example, [5]). The main arguments presented in this paper are:

1. It is profitable to treat the formulæ that describe a pedigree and its associated information as a computer program that can compute the probability of the given information as a function of various parameters of interest (e.g., recombination fractions).

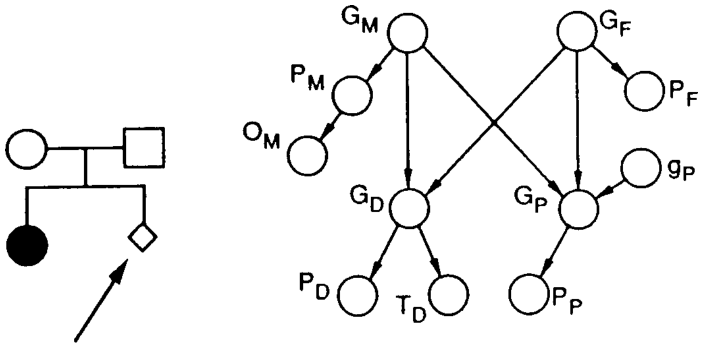
2. A number of typical compiler optimization techniques can be applied to the resulting program to speed up computations, typically at the expense of (much) more space and more compile-time analysis. We examine specifically the roles of *constant folding* and *loop unrolling*.

A.1 Pedigrees as Formulæ

To illustrate pedigree computations, consider the very simple family pedigree shown in Figure A-1a, whose structure as a Bayes network appears in Figure A-1b.[2] An actual problem of clinical or research interest would have many more features, but this simple example should serve to illustrate the issues. For a Bayes network, the joint probability of any complete assignment of values to all the variables is given by the formula:

$$P(x_1, x_2, \ldots, x_n) = \prod_{1 \leq i \leq n} P(x_i | \pi_{x_i}). \tag{A.1}$$

[2]Although it may seem more natural to treat the gender of the propositus as dependent on his or her genotype, it is convenient instead to make the propositus' genotype jointly dependent on the parents' genotypes and the gender. This allows consideration of different modes of inheritance—e.g., autosomal recessive vs. x-linked—simply by altering the conditional probability matrix at this node.



(*a*) (*b*)

Figure A-1: (a) A very simple pedigree, as it might arise in genetic counseling. The couple have had a daughter affected by a suspected Mendelian disorder whose mode of inheritance is not certainly known, and are concerned about the risk to the propositus, who is indicated by the arrow. (b) The pedigree of (a) corresponds to a network of variables interrelated by probabilistic dependencies. M, F, D and P subscripts on nodes correspond to the mother, father, daughter and propositus, respectively. Nodes labeled G stand for the individual's genotype, P for phenotype, g for gender, T for a test whose outcome depends on the genotype, and O for an observation whose outcome depends on the phenotype. In a Bayes network such as this, it is the *absence* of connections that is significant—indicating probabilistic independence.

where π_x is the set of predecessor nodes for x (those on which it depends probabilistically). For x without predecessors, $P(x|\pi_x) = P(x)$ is simply the *a priori* probability.

When (as is typical) only some subset of variables $X' \subseteq X$ have known values, then the joint probability of the set of known values is simply the sum of terms of the form (A.1), one for each combination of possible values of the unknown variables $X \setminus X'$. If we number the variables so that the first $k = |X \setminus X'|$ are unknown, and if V_i is the set of values that the variable x_i may take on, then

$$P(X') = P(x_{k+1}, \ldots, x_n) = \sum_{x_1 \in V_1, x_2 \in V_2, \ldots, x_k \in V_k} P(x_1, x_2, \ldots, x_n). \tag{A.2}$$

Note that if X' is empty, i.e., if no variable has been assigned a value, then the joint probability will be 1.0, summing over all possible states of the world. To calculate the conditional probability of some set of variable assignments X_J given X_I (where I and J represent sets of variables), we use the definition of conditional probability: $P(X_J|X_I) = P(X_J, X_I)/P(X_I)$.

Although conceptually simple, this is not an advantageous computational approach. For example, if in the pedigree of Figure A-1a just the four genotypes and the phenotype of the propositus are unknown, (A.2) will sum over 162 terms, and in general the number of terms will be exponential in the number of unknowns. For families without consanguinity, [3] shows that such formulæ may be factored along the lines of parental descent to form a more efficient, but mathematically equivalent expression. Recently-developed algorithms for general Bayes networks consider a more varied set of factorizations that can also handle consanguinous pedigrees. The basic idea remains the same: The structure of the network shows that many of the variables can be made independent once "in-between" nodes have been instantiated (given definite values); those instantiated nodes are not necessarily those corresponding to the genotypes of the parents, however. For example, our implementation of Cooper's algorithm [2] will derive the formula

$$\begin{aligned} P \;=\; \widetilde{\sum_{P_M,G_D}} \Bigg\{ \widetilde{\sum_{G_F}} \Bigg[& P(G_F) \widetilde{\sum_{g_P}} \Bigg(P(g_P) \widetilde{\sum_{P_P,G_P}} \Big\{ P(P_P|G_P) \widetilde{\sum_{G_M}} [P(G_P|g_P, G_M, G_F) \\ & \qquad \times P(G_D|G_M, G_F) \\ & \qquad \times P(P_M|G_M)P(G_M)] \Big\} \Bigg) \\ & \times \widetilde{\sum_{P_F}} P(P_F|G_F) \Bigg] \\ & \times \Big(\widetilde{\sum_{T_D}} P(T_D|G_D) \Big) \Big(\widetilde{\sum_{P_D}} P(P_D|G_D) \Big) \Big(\widetilde{\sum_{O_M}} P(O_M|P_M) \Big) \Bigg\} \end{aligned} \tag{A.3}$$

for the joint probability of the variables in the network of Figure A-1b, in case no variables are instantiated ($X' = \emptyset$, when of course the expression simply sums to 1.0). Observe that we may in fact use the same expression to evaluate probabilities when some variables are instantiated, by using the following definition of our variant of the summation operator: When a variable x is instantiated, say to the value v, the meaning of $\widetilde{\sum}_x f(x)$ is just $f(v)$, but if it is not instantiated, then $\widetilde{\sum}_x f(x) = \sum_{x \in V(x)} f(x)$. Then, (A.3) subsumes all formulæ of the form of (A.2), but requires substantially less computation. Under the assumption that P_M, P_F, P_D, O_M, T_D, and g_P are known, for example, (A.3) requires roughly one third as many multiplications for its evaluation. Additional savings are realized by a caching strategy which exploits the fact that in evaluating conditional probability formulæ, many components of the calculation of the numerator may be saved and used again in computing the denominator [2]. This point is not further discussed here, but makes a large practical difference in our implementation. Although networks with many undirected cycles (such as G_M, G_D, G_F, G_P, G_M in Figure A-1b and those introduced by consanguinity in pedigrees) in general require exponentially much computation, several factors often help make the computations tractable. Small cycles such as the example here, which result simply from the dependence of all sibling genotypes on the genotypes of their parents, may be eliminated by introducing an extra node for the joint parental genotype [4]; the actual degree of inbreeding in human families is often small; and heuristic factoring methods appear to be practically quite effective, despite the impossibility of theoretical guarantees.

However, there are many situations in which many repeated evaluations of a pedigree are necessary, and methods that are convenient for a single evaluation become unacceptably slow when repeated often. In our application, where information about a family pedigree is presented item-by-item, the pedigree must be repeatedly evaluated to indicate to the counselor the impact of each additional fact. In linkage analyses, inference over a pedigree may have to be done many times to try to estimate the probability of a genetic model being the correct one or attempting to fit a maximum-likelihood formulation. In such cases, additional methods for speeding up the calculation are required. We now turn to consideration of several such methods.

A.2 Pedigree Formulæ as Computer Programs

Often the most successful speed-up methods require some significant insight into the nature of the problem being solved or the solution method. The factorization scheme already exploited here falls into this category. A second general principle of optimization is that any *constant* feature of a problem may be exploited to advantage. This observation underlies most of the methods to be presented here. The third principle is that often one valuable commodity may be traded off for another. In computing, typical tradeoffs are between space and time. We turn to a few specific techniques for exploiting constant features of a pedigree and some possible time/space tradeoffs.

In a typical probabilistic inference program, the expression that describes a probability to be calculated is represented as some data structure in the memory of the computer. The program that performs the computation then traverses that structure, *interpreting* it to determine what detailed computations must actually be done. The work done by the program thus consists of two parts: (a) figuring out the values to be operated on and the exact arithmetic operations to be performed, and (b) actually doing them. When a pedigree with fixed structure is to be evaluated repeatedly, part (a) of this work is still done once for each evaluation—a wasted duplication of effort that we eliminate.

Formula (A.3) may be translated directly into a computer program that, when given an instantiation of some of the variables in the network, computes the joint probability of those instantiations. For example, our translation into LISP[3] of the last line of (A.3) is

```
(* (sum (td) (p td gd))
   (* (sum (pd) (p pd gd)) (sum (om) (p om pm))))
```

where an expression such as `(p td gd)` stands for $P(T_D|G_D)$ and is implemented as an array retrieval, and `*` is a conditional k-ary multiplication that returns zero as soon as any of its arguments is zero (to avoid unnecessary multiplications by zero, which arise often). `Sum` implements $\widetilde{\sum}$: it checks to see if its variable is already instantiated, in which case it simply evaluates its expression; otherwise it sets up a loop to iterate over the valid values of the variable, summing the expression over all instantiations. In fact, each of these forms is further macro-expanded to the primitive LISP expressions that implement these operations. Then, the LISP compiler is invoked to further translate the given program into the machine's primitive instructions set.

In our limited experiments with this technique, it speeds up the calculation of probabilities over pedigrees by a factor of between three and eight, depending on which LISP implementation is being used and specifics of that compiler's own optimizations. Correction of known deficiencies in some of these compilers or the use of different target languages should lead to even greater speed improvements. The only costs of this method are the additional time required for the two compilation steps (formulæ to program, and program to machine language) and the additional space required by the compiled program. Unless these steps are unduly slow, this seems eminently worth doing.

In some series of calculations, we may also know that certain parameters will be held constant. For example, we considered the case above where all variables except the four genotypes and the propositus' phenotype are known. In this case, we may compile a special-purpose version of the general expression (A.3):

```
(sum (gd)
  (* (sum (gf)
       (* (p gf)
          (p GENP)
          (sum (pp)
            (sum (gp)
              (* (p pp gp)
                 (sum (gm)
                   (* (p gp GENP gm gf) (p gd gm gf) (p PM gm) (p gm))))))
          (p PF gf)))
     (p TD gd) (p PD gd) (p OM PM)))
```

[3]Our experiments have thus far been performed in LISP, primarily because it is easy to invoke the LISP compiler from within a running program and then to call the just-compiled function—something that is very difficult or impossible in most programming languages. There is no reason, however, that the methods we describe here should be limited to LISP. F RTRAN, C, or other languages could also be targets, using some sophisticated features of the operating system shell to coordinate the resulting executable code segments. In fact, because commercial compilers for such languages typically generate much better optimized machine code than LISP compilers, there are practical advantages to moving to such a language, at least as the *target* of the system.

Here, upper-cased variables have known constant values and may be replaced by those values, making possible further simplifications of the function. `(p GENP)`, for example, is simply the *a priori* probability of that known value and may be replaced by that constant; it may also be factored out of the loops over values of G_D and G_F. In the factor `(p gp GENP gm gf)`, which corresponds to looking up a value in a rank-four array, the fact that one index is a known constant means that the retrieval can be done over a reduced (rank-three) array, with correspondingly fewer index calculations. If more parameters are known constant, further corresponding simplifications are possible. An almost equivalent way to view this optimization is that the original Bayes net of Figure A-1b may be simplified by graph-reduction methods developed by Shachter [7] when some of the probabilistic nodes are known constant. The expression to compute probabilities for that simpler network may then be factored and translated to an expression similar to that shown above.

Once we know which parameters of a model are actually to be treated as constant, the process described here corresponds to the well-known compiler optimization technique called *constant folding* and its offshoots. The principal difficulty in applying it is to know what can be treated as a constant for enough evaluations of the pedigree formula that it is worth going to the extra effort of compilation to speed up the special case. In our genetic counseling application, this optimization has not yet been used because frequent "what if" calculations have made it difficult to predict which instantiated variables will truly remain fixed with their current values. It should be more useful in applications such as linkage analysis, where known data typically do remain constant.

A final speed-up method exploits a possible tradeoff between the size of the compiled program and its running time. For example, because the variable `gm`, the mother's genotype, is known to take on only a small fixed set of values—for example, $\texttt{GM}_1$, $\texttt{GM}_2$ and $\texttt{GM}_3$, namely "homozygous normal," "heterozygous" and "homozygous affected"—we may expand the expression
`(sum-over (gm) (* (p gp GENP gm gf) (p gd gm gf) (p PM gm) (p gm)))` to

```
(+ (* (p gp GENP GM1 gf) (p gd GM1 gf) (p PM GM1) (p GM1))
   (* (p gp GENP GM2 gf) (p gd GM2 gf) (p PM GM2) (p GM2))
   (* (p gp GENP GM3 gf) (p gd GM3 gf) (p PM GM3) (p GM3))).
```

After such *loop unrolling*, the execution time of the code controlling the loop has been eliminated, and further optimizations (e.g., constant folding) are possible because of the newly-introduced constants.

If applied throughout the program representing a large pedigree, the whole program would be translated into one enormous arithmetic expression consisting of sums of products of sums of products of ..., where the leaf terms would all be constants or references to the parameters of the problem. This structure has a number of significant benefits. For instance, compilers can generally produce very good code for such expressions, and implementation on fine-grained parallel machines can effectively exploit the obvious independence of different parts of the expression to overlap their computations. The major drawback is that such a fully unrolled expression for problems of actual interest might easily contain millions of terms, thus exacting a large space penalty for the improved speed and severely taxing the ability of real compilers to handle such large arithmetic expressions. One avenue of escape from this extreme is to unroll loops only in part, reducing both the speed gain and the space loss. Unrolling the loops furthest inside outer summations will generally make the greatest speed gains at the least space cost, but we have as yet no useful experience to report on how to make that tradeoff. It is also possible that even the extreme form of loop unrolling and the resulting huge expression are an appropriate way to organize computation for scalable supercomputer architectures; indeed, one such experiment has recently been reported for computational dynamics problems [1].

A.3 Discussion

We have reviewed the typical computational approach to calculating likelihoods over pedigrees, emphasizing that the pedigree is effectively translated into an equivalent mathematical formula which may be manipulated to exploit independence. In a similar way, we suggest that such a formula can then be translated into a computer program that computes its value. The resulting program may be further manipulated by techniques much like those applied in compiler optimization to exploit known characteristics (e.g., constants) of the problem and possible tradeoffs of the computing environment.

Optimizations such as these can achieve modest but significant improvements (perhaps an order of magnitude) in the run-time of large problems without introducing significant offsetting penalties. More extreme methods, such as complete loop unrolling, can gain larger improvements, but only at very large costs in space; such a tradeoff may only make sense in the context of some unusual computer architectures.

Ultimately, any optimization that gains a constant factor, no matter how large, will be overwhelmed by the exponential growth of computing cost as we increase the size and complexity of the pedigree or of the genetic model. This will most likely lead to another tradeoff: giving up exact solutions and instead applying sampling techniques based on variants of Monte Carlo simulations. Nevertheless, a significant constant improvement can bring some interesting, slightly larger problems within range of available computational techniques.

[This page intentionally left very nearly blank.]

Bibliography

[1] Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23:25–37, Dec, 1990.

[2] Gregory F. Cooper. Bayesian belief-network inference using nested dissection. Technical Report KSL-90-05, Stanford University, Knowledge Systems Laboratory, Stanford, California, January 1990.

[3] R. C. Elston and A. J. Stewart. A general model for the genetic analysis of pedigree data. *Hum. Hered.*, 21:523–542, 1971.

[4] Nomi Harris. Probabilistic belief networks for genetic counseling. *Computer Methods and Programs in Biomedicine*, Accepted for publication.

[5] K. Lange and R. C. Elston. Extensions to pedigree analysis. *Hum. Hered.*, 25:95–105, 1975.

[6] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[7] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 34:871–882, 1986.

[8] David J. Spiegelhalter. Fast algorithms for probabilistic reasoning in influence diagrams, with applications in genetics and expert systems. In *Conference on Influence Diagrams*, Berkeley, May 1988. University of California.

[This page intentionally left very nearly blank.]

Appendix B

Pedigree Profiling Scheme Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; P_pedigree is a sample cond. prob. pedigree expr. equation             ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; P = SUM_{G_D} { SUM_{G_F} P(G_F)
;;;                            P(GP)  ;<---  GP known  ------------.
;;;                            SUM_{P_P} SUM_{G_P} P(P_P|G_P)  ;   V
;;;                                                SUM_{G_M} P(G_P|GP,G_M,G_F)
;;;                                                          P(G_D|G_M,G_F)
;;;                                                          P(PM|G_M) ; PM known
;;;                                                          P(G_M)
;;;                            P(PF|G_F)  ; PF known
;;;                  P(TD|G_D)  ; TD known
;;;                  P(PD|G_D)  ; PD known
;;;                  P(OM|PM)   ; OM & PM known [hence the P() known]
;;;                }

(define G_M (make-vector 4 (/ 1.0 4)))
(define PM .5)
(define OM .5)

(define G_F (make-vector 4 (/ 1.0 4)))
(define PF .5)

(define G_D (make-vector 4 (/ 1.0 4)))
(define PD .5)
(define TD .5)

(define G_P (make-vector 4 (/ 1.0 4)))
(define gP .5)
(define P_P (make-vector 8 (/ 1.0 8)))

(define (P_pedigree)
  ;;--------------------------------------------------------------------------
  "The dreaded moby pedigree probability of expression from Szolovits' paper."
  ;;--------------------------------------------------------------------------
  (vector-summation G_D                                     ; v-|(|Sigma|)-1.0
    (lambda (g_d)
      (sc-*                                                 ; sc-*-2.0
       (lambda ()
         (vector-summation G_F                              ; v-|(|Sigma|)-3.0
           (lambda (g_f)
             (sc-*                                          ; sc-*-4.0
              (lambda () (P g_f))                           ;    P-4.1
              (lambda () (P gP))                            ;    P-4.2
              (lambda ()
                (vector-summation P_P                       ; v-|(|Sigma|)-5.0
                 (lambda (p_p)
                   (vector-summation G_P                    ; v-|(|Sigma|)-6.0
                     (lambda (g_p)
                       (sc-*                                ; sc-*-7.0
                        (lambda () (P p_p g_p))             ;    P-7.1
                        (lambda ()
                          (vector-summation G_M             ; v-|(|Sigma|)-8.0
                            (lambda (g_m)
                              (sc-*                         ; sc-*-9.0
                               (lambda () (P g_p gP g_m g_f)) ;    P-9.1
                               (lambda () (P g_d    g_m g_f)) ;    P-9.2
                               (lambda () (P PM     g_m    )) ;    P-9.3
                               (lambda () (P        g_m    )) ;    P-9.4
                               ))))))))))
              (lambda () (P PF g_f))))))                    ;    P-4.4
       (lambda () (P TD g_d))                               ;    P-1.2
       (lambda () (P PD g_d))                               ;    P-1.3
       (lambda () (P OM PM))))))                            ;    P-1.4
```

```
;;; Pedigree utility subroutines

(define (vector-summation v proc)
  ;;-------------------------------------------------------------------------
  "(VECTOR-SUMMATION v proc) sums the invocation of PROC on each vector V elt."
  ;;-------------------------------------------------------------------------
  (do ((index (-1+ (vector-length v)) (-1+ index))
       (acc             0            (+ acc (proc (vector-ref v index)))))
      ((negative? index) acc)))

(define (sc-* . thunks)
  (string-append
  ;;------------------------------------------------------------------------
   "(SC-* . thunks) short-circuit times of thunk calls from left-to-right."    newline-string
   "If any thunk call yields 0, short-circuits to return 0 w/o calling thunks" newline-string
   "to its right.")
  ;;------------------------------------------------------------------------
  (do ((thunks-2-do thunks (cdr thunks-2-do))
       (acc           1    (* acc ((car thunks-2-do)))))
      ((or (null? thunks-2-do)
           (zero? acc))
       acc)))

(define (P val . givens)
  ;;------------------------------------------------------------------------------------
  "(P val . givens) computes the conditional probability of VAL given all GIVENS values."
  ;;------------------------------------------------------------------------------------
  ;; This would actually do a table lookup in a table of conditional probabili-
  ;;  ties. To keep things simple for now, we just effectively return VAL.
  (let ((given-val (reduce * 1 givens)))
    (/ (* val given-val)
       given-val)))
```

```
;;; Prtotype profiling...

(define *sample-pedigree-profiling-table*
  '(;;;(proc-name . nesting-depth)
    (v-Sigma-1.0 . 1)
    (---sc-*-2.0 . 2)
    (v-Sigma-3.0 . 3)
    (---sc-*-4.0 . 4)
    (---P----4.1 . 4)
    (---P----4.2 . 4)
    (v-Sigma-5.0 . 5)
    (v-Sigma-6.0 . 6)
    (---sc-*-7.0 . 7)
    (---P----7.1 . 7)
    (v-Sigma-8.0 . 8)
    (---sc-*-9.0 . 9)
    (---P----9.1 . 9)
    (---P----9.2 . 9)
    (---P----9.3 . 9)
    (---P----9.4 . 9)
    (---P----4.4 . 4)
    (---P----1.2 . 1)
    (---P----1.3 . 1)
    (---P----1.4 . 1)
    (---P----0.0 . 1)
    ))

(define (sample-pedigree/hybrid-profile)
  (for-each (lambda (proc-name+nesting-depth)
              (hybrid-profile/profile-proc (car proc-name+nesting-depth)
                                           (cdr proc-name+nesting-depth)))
            *sample-pedigree-profiling-table*))

(define (profiling-table)
  *sample-pedigree-profiling-table*)
```

```
;;; Fully instrumented equation...

(define (profiled-P_pedigree)
  ;;--------------------------------------------------------------------------
  "The dreaded moby pedigree probability of expression from Szolovits' paper."
  ;;--------------------------------------------------------------------------

  (define V vector-summation)
  (define ($       proc-name thunk)
    (hybrid-profile proc-name thunk))

  ($ 'v-Sigma-1.0                                                                 ; [v-Sigma-1.0]
     (lambda ()
       (V G_D
          (lambda (g_d)
            ($ '---sc-*-2.0                                                       ; [---sc-*-2.0]
               (lambda ()
                 (sc-*
                  (lambda ()
                    ($ 'v-Sigma-3.0                                               ; [v-Sigma-3.0]
                       (lambda ()
                         (V G_F
                            (lambda (g_f)
                              ($ '---sc-*-4.0                                     ; [---sc-*-4.0]
                                 (lambda ()
                                   (sc-*
                                    (lambda () ($ '---P----4.1                    ; [---P----4.1]
                                                  (lambda ()
                                                    (P g_f))))
                                    (lambda () ($ '---P----4.2                    ; [---P----4.2]
                                                  (lambda ()
                                                    (P gP))))
                                    (lambda ()
                                      ($ 'v-Sigma-5.0                             ; [v-Sigma-5.0]
                                         (lambda ()
                                           (V P_P
                                              (lambda (p_p)
                                      ($ 'v-Sigma-6.0                             ; [v-Sigma-6.0]
                                         (lambda ()
                                           (V G_P
                                              (lambda (g_p)
                                                ($ '---sc-*-7.0                   ; [---sc-*-7.0]
                                                   (lambda ()
                                                     (sc-*
                                                      (lambda ()
                                                        ($ '---P----7.1           ; [---P----7.1]
                                                           (lambda ()
                                                             (P p_p g_p))))
                                                      (lambda ()
                                                        ($ 'v-Sigma-8.0           ; [v-Sigma-8.0]
                                                           (lambda ()
                                                             (V G_M
                                                                (lambda (g_m)
                                                                  ($ '---sc-*-9.0 ; [---sc-*-9.0]
                                                                     (lambda ()
                                                                       (sc-*
                                                                        (lambda ()
                                                                          ($ '---P----9.1 ; [---P----9.1]
                                                                             (lambda ()
                                                                               (P g_p gP g_m g_f))))
                                                                        (lambda ()
                                                                          ($ '---P----9.2 ; [---P----9.2]
                                                                             (lambda ()
                                                                               (P g_d    g_m g_f))))
                                                                        (lambda ()
                                                                          ($ '---P----9.3 ; [---P----9.3]
                                                                             (lambda ()
                                                                               (P PM     g_m    ))))
                                                                        (lambda ()
                                                                          ($ '---P----9.4 ; [---P----9.4]
                                                                             (lambda ()
                                                                               (P        g_m    ))))
                                                                        ))))))))))))))))))))
```

```
                                    (lambda () ($ '---P----4.4                    ; [---P----4.4]
                                                  (lambda ()
                                                    (P PF g_f)))))))))))))
                  (lambda () ($ '---P----1.2                                      ; [---P----1.2]
                                (lambda ()
                                  (P TD g_d))))
                  (lambda () ($ '---P----1.3                                      ; [---P----1.3]
                                (lambda ()
                                  (P PD g_d))))
                  (lambda () ($ '---P----1.4                                      ; [---P----1.4]
                                (lambda ()
                                  (P OM PM)))))))))))))

(define (go)
  ( hybrid-profile/reset)
  (sample-pedigree/hybrid-profile)
  (      pc-sample/start-sampling-thread)

  (timed profiled-P_pedigree)

  (     pc-sample/stop-sampling-thread)
  (hybrid-profile/displayable-samplings)
  )

;;; fini

;;;(provide "Sample-- Conditional Probability of Pedigree Expression")
```

Appendix C

Prototype Program Profiler Scheme Code

```
;;; Primitive [crude] profiling

;;;(require "List Utilities"             "list.scm")  ; For index-list
;;;(require "Optional Arguments" "optional-args.scm")
;;;(require "Stack Queues"        "stack-queue.scm")

(define (profile-proc/register-proc profile-proc-registery proc-name . nesting-depth)
  ;; NB: profile-proc-registery assumed to be an alist keyed on proc-name
  (let ((frame
         (profile-proc/lookup-proc proc-name profile-proc-registery)))
    (cond ((found? frame)
           (error "Proc already being PC sampled" proc-name))
          (else
           (set-cdr! profile-proc-registery
                     (cons (apply profile-proc/make-frame proc-name nesting-depth)
                           (cdr profile-proc-registery)))
           'done))))

(define (profile-proc/lookup-proc proc-name profile-proc-registery)
  (assq proc-name (cdr profile-proc-registery)))

(define (profile-proc/sort-registery profile-proc-registery)
  (let ((header (car profile-proc-registery)))
    (cons header
          (sort (cdr profile-proc-registery)
                (lambda (frame-1 frame-2)
                  (> (profile-proc-frame/weighted-count header frame-1)
                     (profile-proc-frame/weighted-count header frame-2)))))))

(define (profile-proc-frame/weighted-count profile-registery-header
                                           profile-proc-frame)
  (* (profile-proc-frame/count-weight      profile-registery-header
                                           profile-proc-frame)
     (profile-proc-frame/count             profile-proc-frame)))

(define (profile-proc-frame/count-weight profile-registery-header
                                         profile-proc-frame)
  (if (found? (memq profile-registery-header
                    '(---pc-samplings---
                      -trace-profiling--)))
      1
      (nesting-weight
       (profile-proc-frame/nesting-depth profile-proc-frame))))

(define (   identity-nesting-weight w)    w   )
(define (     linear-nesting-weight w) (* w 10))
(define (     square-nesting-weight w) (* w  w))
(define (       poly-nesting-weight w) (+ (square-nesting-weight w)
                                          (linear-nesting-weight w)))
(define (      cubic-nesting-weight w) (* w w w))
(define (  factorial-nesting-weight w) (factorial w))
(define (  quadratic-nesting-weight w) (expt w *quad-expt*))
(define (  fibonacci-nesting-weight w) (fib w))
(define (natural-exp-nesting-weight w) (exp w))
(define (    PI-expt-nesting-weight w) (expt pi w))
(define (double-expt-nesting-weight w) (expt w w))

(define *quad-expt* 4)

(define (nesting-weight w) (double-expt-nesting-weight w))
```

```
;;; Profiling Registery... the backbone profiling data structure

(define (profile-proc-registery/percentilate-registery profile-proc-registery)
  (let ((sorted-samplings
         (profile-proc/sort-registery profile-proc-registery))
        (header (car profile-proc-registery))
        (count-tally
         (let ((header (car profile-proc-registery)))
           (do ((frames (cdr profile-proc-registery)
                        (cdr frames))
                (acc    0
                        (+   acc
                             (profile-proc-frame/count        (first frames))))
                (w-acc  0
                        (+ w-acc
                           (profile-proc-frame/weighted-count header
                                                              (first frames)))))
               ((null? frames)
                (cons (/   acc 100.0)
                      (/ w-acc 100.0)))))))
    (define count-tally/normalized-total          car)
    (define count-tally/normalized-weighted-total cdr)

    (define (profile-proc-registery/percentilate-profile-proc-frame frame)
      (profile-proc-frame/append
         frame
         (let ((weighted-count (profile-proc-frame/weighted-count header
                                                                  frame)))
           (list (/ (profile-proc-frame/count                     frame)
                    (count-tally/normalized-total         count-tally ))
                 weighted-count
                 (/ weighted-count
                    (count-tally/normalized-weighted-total count-tally ))))))
    (cons (car sorted-samplings)
          (map profile-proc-registery/percentilate-profile-proc-frame
               (cdr sorted-samplings)))))

(define (profile-proc-registery->displayable-registery profile-proc-registery)
  (let ((percentilated-registery
         (profile-proc-registery/percentilate-registery profile-proc-registery)))
    (let ((indexed-percentilated-registery
           (cons (car percentilated-registery)
                 (index-list (cdr percentilated-registery) 0))))
      indexed-percentilated-registery)))


(define (profile-proc/make-frame proc-name . nesting-depth)
  (if (optional-argument-supplied? nesting-depth)
      (set! nesting-depth (optional-argument-contents nesting-depth))
      (set! nesting-depth 1))
  (list proc-name nesting-depth 0))     ; invoked 0 times so far

(define  profile-proc-frame/append        append)
(define  profile-proc-frame/proc-name     first)
(define  profile-proc-frame/nesting-depth second)
(define  profile-proc-frame/count         third)
(define (profile-proc-frame/incr-count! frame . incr)
  (if (optional-argument-supplied? incr)
      (set! incr (optional-argument-contents incr))
      (set! incr 1))
  (set-cdr! (cdr frame) (list (+ incr (profile-proc-frame/count frame)))))


(define *current-running-proc* '<idle-fnord>)
```

```
;;; PC Sampling [crude]

(define *pc-sampling-period*     1)     ; milliseconds
(define *pc-samplings-registery* (list '---pc-samplings---))

(define (pc-sample/reset)
  (cond ((pc-sampling-thread-installed?)
         (pc-sample/stop-sampling-thread)))
  (set! *pc-samplings-registery* (list '---pc-samplings---)))

(define (pc-sample/profile-proc     proc-name . nesting-depth)
  (apply profile-proc/register-proc
         *pc-samplings-registery*   proc-name   nesting-depth))

(define *pc-sampling-thread* 'to-be-installed---by-invoking--pc-sample)
(define (pc-sampling-thread-installed?)
  (thread? *pc-sampling-thread*))

(define (pc-sample/start-sampling-thread)
  (set! *pc-sampling-thread*
        (create-thread
         false                          ; no continuation
         (lambda ()
           (let loop ()
             (cond ((not (eq? *current-running-proc* '<idle-fnord>))
                    (let ((current-proc-frame
                           (profile-proc/lookup-proc *current-running-proc*
                                                     *pc-samplings-registery*)))
                      (cond ((not-found? current-proc-frame)
                             (error "PC-SAMPLE-THREAD sampled a vacuum"
                                    proc-name)))
                      (profile-proc-frame/incr-count! current-proc-frame)))
                   (else
                    (newline)
                    (display "Idle pc sample")))
             (sleep-current-thread *pc-sampling-period*)
             (loop)
             ))))
  (detach-thread *pc-sampling-thread*)  ; no joins
  *pc-sampling-thread*)

(define (pc-sample/stop-sampling-thread)
  (cond ((not (thread-dead? *pc-sampling-thread*))
         (signal-thread-event *pc-sampling-thread*
                              (lambda () (exit-current-thread 'reset))))))


(define (pc-sample/displayable-samplings)
  (profile-proc-registery->displayable-registery *pc-samplings-registery*))
```

```
;;; Flow profiling [crude]

(define *flow-profiling-registery* (list '--flow-profiling--))

(define (flow-profile/reset)
  (set! *flow-profiling-registery* (list '--flow-profiling--)))

(define (flow-profile/profile-proc  proc-name . nesting-depth)
  (apply profile-proc/register-proc
         *flow-profiling-registery* proc-name   nesting-depth))

(define (with-flow-profiled-proc/checkin proc-name thunk)
  ;;; Sigh? Cannot use fluid let since would un-let the current proc every
  ;;;       time the pc-sampling thread kicks in.
  (let ((old-proc *current-running-proc*))
    (set! *current-running-proc* proc-name) ; for pc sampling
    (let ((result (thunk)))
      (set! *current-running-proc* old-proc)
      result)))

(define (with-timed-thunk-call thunk exit)
  (let ((start-time (get-profiling-time)))
    (let ((result (thunk)))
      (let ((done-time (get-profiling-time)))
        (exit result (- done-time start-time))))))

(define (get-profiling-time)
  ;; Might some day wanna use (real-time-clock) instead for kicks.
  (runtime))

(define (flow-profile proc-name thunk)
  (with-flow-profiled-proc/checkin proc-name
    (lambda ()
      (with-timed-thunk-call thunk
        (lambda (result time)
          (let ((frame (profile-proc/lookup-proc *current-running-proc*
                                                 *flow-profiling-registery*)))
            (cond ((not-found? frame)
                   (error "PROFILE tried to sample a non-beingness"
                          proc-name)))
            (profile-proc-frame/incr-count! frame time)
            result))))))

(define (flow-profile/displayable-samplings)
  (profile-proc-registery->displayable-registery *flow-profiling-registery*))
```

```
;;; Tracing [crude]

(define *trace-profiling-registery* (list '-trace-profiling--
                                          (make-stack-queue)))
(define (trace-profile/reset)
  (stack-queue-reset! (second *trace-profiling-registery*)))

(define (trace-profile proc-name thunk)
  (with-trace-profiled-proc/checkin proc-name
    (lambda ()
      (thunk))))

(define (with-trace-profiled-proc/checkin proc-name thunk)
  (let ((start-time (get-profiling-time)))
    (let ((result (thunk)))
      (let ((done-time (get-profiling-time)))
        (stack-queue-enqueue! (cons proc-name (- done-time start-time))
                              (second *trace-profiling-registery*))
        result))))

(define (trace-proc-registery->displayable-registery trace-proc-registery)
  (cons (first trace-proc-registery)
        (stack-queue-contents (second trace-proc-registery))))

(define (trace-profile/displayable-samplings)
  (trace-proc-registery->displayable-registery *trace-profiling-registery*))
```

```
;;; Hybrid profiling... [all of the above... and twice the price]

(define (hybrid-profile/profile-proc proc-name . nesting-depth)
  (apply      pc-sample/profile-proc proc-name   nesting-depth)
  (apply   flow-profile/profile-proc proc-name   nesting-depth)
  'ok)

(define (hybrid-profile/reset)
  (    pc-sample/reset)
  ( flow-profile/reset)
  (trace-profile/reset)
  'reset)

(define (hybrid-profile          proc-name thunk)
  (trace-profile                 proc-name
                 (lambda ()
                   (flow-profile proc-name thunk))))

(define (hybrid-profile/displayable-samplings)
  (let ((=====================================
         (make-string 76 #\=)))
  (list
   =====================================
   (    pc-sample/displayable-samplings)
   =====================================
   ( flow-profile/displayable-samplings)
   =====================================
   (trace-profile/displayable-samplings)
   =====================================
   )))


;;; A Hack, A Tweak, and a Frob [a.k.a. a horse, a bucket, and a spoon]

(define pi 3.141592649)

(define (fib n)
  (if (negative? n)
      0
      (do ((count-down n (-1+ count-down)   )
           (fib-n-1    0 (+ fib-n-1 fib-n-2))
           (fib-n-2    1    fib-n-1         ))
          ((zero? count-down) fib-n-1))))

(define (factorial n)
  (do ((count-down n (-1+ count-down))
       (acc        1 (* acc count-down)))
      ((zero? count-down) acc)))


;;; fini

;;;(provide "Prototype Profiling")
```

[This page intentionally left very nearly blank.]

Bibliography

[Abelson *et al.* 89]
Harold Abelson, Michael Eisenberg, Matthew Halfant, Jacob Katzenelson, Elisha Sacks, Gerald J. Sussman, Jack Wisdom, and Kenneth Yip
Intelligence in Scientific Computing
CACM Vol. 32, No. 5, May 1989, pp.546–562

[Adams 90]
Stephen Adams
Towards Language-Oriented Programming
CSTR 90-5, Feb 1990.
Univeristy of Southampton, Department of Electronics and Computer Science

[Agarwal, Chaiken & Kranz 91]
Anant Agarwal, David Chaiken and David Kranz
Multiprocessor Address Tracing and Performance Analysis
MIT/LCS/TM-446, Apr 1991

[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

[Apple 92]
Apple Computer, Eastern Research and Technology
DYLAN*: An Object-Oriented Dynamic Language*
Apple Computer, Inc, 1992

[Baier & Sager 76]
Jean-Loup Baier and Gary R. Sager
Dynamic Improvement of Locality in Virtual Memory Systems
IEEE Transactions on Software Engineering, Vol. SE2, No. 1, Mar 1976, pp.54–62

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

[Bell 73]
James R. Bell
Threaded Code
CACM Vol. 16 No. 6, Jun 1973, pp.370–372

[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

[Chambers & Ungar 89]
Craig Chambers and David Ungar
Customization: Optimizing Compiler Technology for SELF*, a Dynamically-Typed Object-Oriented Programming Language*
In Proc. ACM PLDI 89, Jun 1989, pp.146–160

[Chambers & Ungar 90]
Craig Chambers and David Ungar
Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs
In Proc. ACM PLDI 90, Jun 1990, pp.150–164

[Chambers & Ungar 91]
Craig Chambers and David Ungar
Making Pure Object-Oriented Languages Practical
In Proc. ACM OOPSLA 91, May 1991, pp.1–15

[Chambers, Ungar & Lee 89]
Craig Chambers, David Ungar and Elgin Lee
An Efficient Implementation of SELF*, a Dynamically-Typed Object-Oriented Language Based on Prototypes*
In Proc. ACM OOPSLA 89, Oct 1989, pp.49–70

[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.

[Cooper 90]
Gregory F. Cooper
Bayesian Belief-Network Inference Using Recursive Decomposition
Report No.KSL 90-05, Stanford Knowledge Systems Laboratory, Medical Computer Science, Feb 8, 1990

[Dershowitz 83]
Nachum Dershowitz
The Evolution of Programs
Progress in Computer Science No. 5, Birkhäuser Boston, Inc., 1983

[Driscoll & Orth 86]
Graham C. Driscoll, Jr. and Donald L. Orth
Compiling APL*: The Yorktown* APL *Translator*
IBM Journal of Research and Development Vol. 30, No. 6, Nov 1986, pp.583–593

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

[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

[Harper & Mitchell 87]
Robert Harper and Kevin Mitchell
Introduction to STANDARD ML
Computer Science Department, University of Edinburgh, Sept 1986

[Hayes-Roth 88]
Barbara Hayes-Roth
Making Intelligent Systems Adaptive
Stanford STAN-CS-88-1126 and/or KSL-88-64, Oct 1988

[Hölzle, Chambers & Ungar 91]
Urs Hölzle, Craig Chambers and David Ungar 91]
Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches
In Proc. ECOOP 91, Jul 1991, pp.21–38, Springer-Verlag, Lecture Notes in Computer Science No. 512

[Hudak, Wadler, *et al.* 88]
Paul Hudak, Philip Wadler, *et al.*
Report on the Programming Language HASKELL*: Draft Proposed Standard*
Yale University Department of Computer Science, YALEU/DCS/RR-666, Dec 1988

[Johnston 79]
Ronald L. Johnston
The Dynamic Incremental compiler of APL*3000*
In Proc. ACM APL 79, APL Quote Quad Vol. 9, No. 4-Part 1, Jun 1979

[Keppel 91]
David Keppel
A Portable Interface for On-The-Fly Instruction Space Modification
In Proc. ACM ASPLOS 91, Apr 1991, pp.86–95

[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

[Kesselman 91]
Carl Kesselman
Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs
Ph.D. Dissertation, UCLA CSD-910015, Jun 1991

[Kyrimis & Alonso 88]
Kriton Kyrimis and Rafael Alonso
An Experimental Comparison of Initial Placement vs. Process Migration for Load Balancing Strategies
Princeton University CS-TR-199-88, Dec 1988

[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

[Lumer 91]
Erik Lumer
Ultradiffusion Algorithm for Resource Management In Distributed Systems
Xerox PARC SSL-91-103, System Sciences Laboratory, Xerox Palo Alto Research Center, 1991

[Massalin & Pu 90]
Henry Massalin and Calton Pu
Fine-Grain Adaptive Scheduling using Feedback
USENIX Computing Systems, Vol. 3, No. 1, Winter 1990, pp.139–173

[Milner 84]
Robin Milner
A Proposal for STANDARD ML
In Proc. 1984 ACM Symposium on LISP and Functional Programming, Aug 1984, pp.184–197

[Mitchell 81]
Tom M.Mitchell
Generalization as Search
In Readings in Artificial Intelligence, Bonnie Lynn Webber and Nils J.Nilson (Editors) Morgan Kaufman Publishers, Inc., 1981, pp.517–542

[Pettis & Hansen 90]
Karl Pettis and Robert C. Hansen
Profile Guided Code Positioning
In Proc. ACM PLDI 90, Jun 1990, pp.16–27

[Pu, Massalin & Ioannidis 88]
Calton Pu, Henry Massalin and John Ioannidis
The Synthesis Kernel
USENIX Computing Systems, Vol. 1, No. 1, Winter 1988, pp.11–32

[Reilly 90]
Matthew H. Reilly
A Performance Monitor for Parallel Programs
Academic Press Inc., 1990

[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

[Ruf & Weise 91]
Erik Ruf and Daniel Weise
Using Types to Avoid Redundant Specialization
Stanford FUSE MEMO 91-5, a corrected version of Proc. ACM PEPM 91, Jun 91, pp.321-333

[Rustin 72]
Randall Rustin, ed.
Design and Optimization of Compilers
Proc. Courant Computer Science Symposium 5, Mar 1971, Prentice-Hall, Inc., 1972

[Ryder 74]
K. D. Ryder
Optimizing program placement in virtual systems
IBM Systems Journal Vol. 13, No. 4, 1974

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

[Shoch & Hupp 82]
John F. Shoch and Jon A. Hupp
The "Worm" Programs—Early Experience with a Distributed Computation
CACM Vol. 25, No. 3, Mar 1982, pp.172–180

[Stroustrup 86]
Bjarne Stroustrup
The C++ *Programming Language*
Addison-Wesley, 1986

[Szolovits 92]
Peter Szolovits
Compilation for fast calculation over pedigrees
Proc. Genetic Analysis Workshop 7, in Cytogenetics and Cell Genetics 59, pp.136–138, 1992
S. Karger Medical and Scientific Publishers

[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