Chapter 0

Overview

To be interested in the changing seasons is a happier state of mind than to be hopelessly in love with spring.

— GEORGE SANTAYANA (1863–1952)

Chapter Goals

The goal of this chapter is to restate the abstract with sufficient detail, discussion and supporting examples to demonstrate the proof of concept of the thesis statement while explaining the original contributions. By the end of this chapter, the reader should understand the core problems addressed, appreciate why they are important, and be convinced, at least superficially, that they have been resolved in the DESCARTES prototype, both effectively and efficiently.

It is also hoped that the reader will walk away from this chapter wanting to know more while still believing that the overall DESCARTES design is not only plausible but also both simple and elegant. "If wishes were horses, ..."

0.0 Thesis Statement

Dynamically adaptive, profile-driven, spectral program specialization is an effective and practical technique for building large, sophisticated, efficient software systems from high-level, abstract, modular programs using type-driven polyvariant on-line partial evaluation grounded in statistical inference.

0.1 Introduction

A central goal of software engineering is to build ever more powerful software systems. As such systems grow in functionality, they also tend to grow in size, and thus in complexity. To mitigate this, software designers rely on high-level, abstract, modular programming techniques to control this complexity and manage the resulting code.

Unfortunately, these techniques come at a price: large high-level, abstract, modular software systems tend to be slower than their (even larger) low-level, concrete, monolithic equivalents. Program specialization can help by selectively reducing critical sections of high-level, abstract, modular code into lower-level, concrete, monolithic subsystems. Yet, this is exactly the sort of meticulous tedium to which machines are well suited, and humans are not. Performance profiling can help to focus these efforts, although instrumenting and profiling code can lead to additional overhead.

In profiling-based methods of program optimization, one gathers dynamic information from sample program runs in order to focus optimization efforts on those code fragments that stand to benefit most. This ordinarily requires a programmer-crafted suite of program inputs from which dynamic statistical data about a program's behavior can be deduced. Moreover, this profiling is usually done only once and, thereafter, the optimized code is frozen.

Unfortunately, improving a program's *average* performance on some *typical* inputs does not ensure *optimal* performance on any *specific* input. Worse, it is often difficult to ascertain reliably what constitutes a typical input for highly general programs. For example: What is a typical circuit for a circuit simulator? What constitutes a typical program for a compiler or interpreter or operating system? The answers can vary wildly among different sites and program users. They can even vary dramatically from day to day for a user with changing needs and goals.

This is not to say, though, that there do not exist situations where a sufficiently large sample size ultimately converges adequately to characterize typical patterns of usage— for instance, booting a modern operating system takes many millions of instructions, which can constitute a fairly accurate predictor of the mixture and distribution of underlying micro-architectural machine instructions one is likely to encounter over the course of running a typical program on that operating system. This, however, is not a case of workload characterization driven by data diversity so much as it's an instance of diverse instruction set coverage. Knowing *what* is done with *what* frequency does not lend insight into *why* it's done if one simply profiles instructions without also profiling the data that drives those instructions. Programs that are not driven by data are straightforward, and therefore not very interesting from a specialization standpoint.

A program without a loop and a structured variable isn't worth writing.

— ALAN J. PERLIS (1922–1990), *Epigrams on Programming*, No. 18

What is needed is a mechanism for programs to adapt themselves automatically to their diverse and changing application environments.

To that end, the DESCARTES system marries a set of run-time profiling tools to a profile-driven kernel-level source-to-source program specializer and an optimizing compiler for the MIT C SCHEME dialect of LISP. This yields on-the-fly identification and specialization of performance-critical high-level, abstract program modules, selectively reducing them to streamlined equivalent code in a robust, disciplined, judicious and efficient manner.

After the following short digression on the relevance of the problem being addressed, the remainder of this chapter is a high-level overview of this approach. Think of it as "the dissertation in one chapter".

0.2 Motivation: The Domain of Interest

Before going further, let's first more clearly define what is meant by the phrase "high-level, abstract, modular programs". This will illuminate the nature of the specific problem DESCARTES proposes to solve with dynamic program specialization.

0.2.0 What is High-Level, Abstract, Modular Programming?

I have in mind here (at least) three fundamental notions when I use the term *high-level, abstract, modular programming* (or *HAM*, if you will):

1) *Higher-Order Procedures* - procedures that operate on other procedures.

Those in the mathematics community might call these *operators* while logicians might prefer the term *combinators* or *functionals*. Passing procedures as inputs to other procedures and returning them as output enables the programmer to express abstract process structure without having to duplicate patterns of code, thus enhancing maintainability as well as clarity of expression.

Anyone who has implemented an "*iterator*" to invoke a procedure argument on each of a set of data has defined a *higher-order procedure*. Functional language adherents might call such a procedure a *mapper* while theorists might call it a *(covariant) functor*. Either way, such constructs allow the programmer to abstract away the details of any particular iteration from the core objective computation being performed on each point. Examples include vector scaling operators, point-wise list combiners, and such.

Another familiar example might be a general `sort` procedure that takes two arguments, a set of data of some type and a comparison predicate that imposes a partial order on elements of the set, and sorts the input data set with respect to the partial order. For instance, our `sort` would work both on sets of integers with `integer-less` and on sets of reals with `real-less`. It would also work on sets of strings using `string-less` to sort them into lexicographical order. In fact, it would sort any kind of widget on which a `widget-less` procedure is defined. In this way, it allows the programmer to express an algorithm for sorting partially ordered items by abstracting out the details of what kinds of items are being sorted and what comparator to use for pair-wise ordering them. The ubiquitous quicksort (`qsort`) and binary search (`bsearch`) utilities of the POSIX library are classic concrete examples.

Finally, one might wish to abstract over entire families of procedures by defining generating functions to selectively enumerate elements of the family. For instance, one might define a `doubler` to map over a set of numbers multiplying each by 2 or a `halver` to do the same with respect to 1/2. But what one might really want is a higher-order `scaler` functor that takes a scaling argument (like 2 or 1/2) as input and generates the corresponding instance of the broader family of scaling *procedures* (like our `doubler` or `halver` procedures). Note that in this example, the `scaler` functor being defined is dubbed `higher-order` not because it takes a procedure argument as input (it takes a number) but because it returns a (new) procedure as output, one that itself takes as input a set of numbers and scales them appropriately.

In the general case, this requires treating procedures as first-class data objects that are closed over their definition environments. Not surprisingly, therefore, the object-oriented community calls these *objects* while those in the LISP/SCHEME community might prefer the term *full closures*, or just *closures*.[0]

This illustrates the principle of *procedural control abstraction* in which the procedures passed as parameters and/or returned as values constitute abstracted-out pieces of run-time process control.

2) *Generic Procedures* - procedures that operate on a variety of types of data.

Those in the object-oriented programming community might call these *methods* while type theorists might refer to this as an instance of *polymorphism*. For the dynamically typed language used in the DESCARTES prototype, *generic procedures* are implemented by a combination of run-time type conversion followed by type dispatch. For instance, the generic addition procedure adds integers to real numbers by converting the integer into a real then dispatching to the non-generic real addition core procedure.

Writing code in terms of generic procedures rather than their concrete, low-level counterparts makes large systems concise and flexible, and therefore extensible, by encouraging code reuse, as we shall see below. This enhances code maintainability by avoiding re-implementing the same program idiom multiple times.

For instance, one can define a single generic numeric `square` procedure in terms of generic `multiply` rather than having to implement N different versions for N different numeric types, and then being forced to explicitly choose among them at every would-be call site. Worse, for two-argument generic arithmetic procedures, like `sum-of-squares`, N^2 versions would be required, one for each distinct permutation of argument types. In general, this problem grows exponentially with the number of parameters since it grows proportional to their cross product, not to their sum.

The implications can be even worse for code that wants to call what would otherwise have been a generic procedure, seducing implementors into concessions like forced dichotomies: implementing two distinct versions of entire numeric library packages, one for integers and one for reals. By not subsequently supplying generic convert-and-dispatch interfaces, clients of such dichotomous packages are forced to perform their own type conversions explicitly at each call site where conversion is needed before invocation.[1]

This approach becomes worse still when a new concrete data abstraction is added to the mix. For instance, imagine we want to now add a third kind of numeric data, like complex

[0] Of course, *objects*, in the general OOPS sense, also connote encapsulation and other mechanisms. It is not my intent to conflate *closures* with *objects per se*, but only to suggest that at the heart of each OOPS *object* lies a *closure* [Stein 87].

[1] Numerical analysts may well object that this is appropriate, since accumulation of error in IEEE floating-point conversion must be considered carefully for well-behaved results [Martel 2006]. I picked integers and reals as an example only for familiarity. A less controversial (but less commonplace) choice might have been integers and rationals, or `fixnums` and `bignums` (fixed-size vs. arbitrary-precision integers).

numbers. We would first implement concrete base procedures (add, subtract, multiply, *etc.*) on complex numbers, then define conversion procedures to convert from integers and reals to complex numbers (with null imaginary components) and back again (by extracting the real and imaginary subcomponents).

But if we stop there then every other procedure already defined in terms of integer or real arithmetic that we want to extend to operate on complex numbers would have to be re-implemented to extend them over complex numbers. Moreover, every procedure with a separate integer and real implementation might now call for a third version for complex numbers. Entire dichotomous numeric libraries would now become trichotomous.

By contrast, by simply extending existing generic convert-and-dispatch interfaces to handle complex numbers appropriately— *e.g.,* by promoting integers and reals to complex numbers with null imaginary components then dispatching to the concrete complex number base routines— all existing code implemented in terms of generic procedures would be instantly extended to handle complex numbers *without having to re-write or re-implement any other code*. This is a tremendous advantage for serious scientific computation. This, for example, is the approach advocated in [Abelson & Sussmans 96] (from which much of this argument is borrowed). Alas, this advantage can also become a liability when efficiency concerns are paramount, as I will address shortly.

This illustrates what might be called *procedural data abstraction* in that the specific kinds of data on which a procedure operates can be abstracted out, and therefore textually suppressed at their call sites. This allows one to layer generic procedure upon generic procedure, thereby directly expressing general abstract high-level programming concepts while ignoring concrete data conversions and concrete base procedures until the genericity bottoms out in concrete base implementation modules at the library interface boundaries.

Contrast this *procedural data abstraction* with the *procedural control abstraction* afforded by *higher-order procedures* of the previous item.

3) *Modular Procedures* - small procedures that embody single, simple tasks.

This is more a matter of programmer discipline than the preceding two.

The discipline of building *modular procedures* simply refers to breaking down large programs and data abstractions into smaller ones, naming the parts, then hierarchically building them back up from named subcomponents. It also refers to designing several small, simple, independent procedures that can be composed into more complex forms rather than writing large, monolithic procedures that express the complex composition of separable concepts directly in one giant interconnected tangle of code.

Examples include concise library routines and personal utility toolkits.

Small general programs can be combined in multifaceted ways to build up larger, more complex systems and subsystems of code. This systematic decomposition of large tasks into smaller subtasks manages code size complexity. Meanwhile, factoring complex tasks into separate distinct self-contained ideas then re-composing them hierarchically manages conceptual complexity through the principle of orthogonality of design. This enhances code maintainability through unit testing by isolating potential errors to small textually independent code regions.

This illustrates what might be called *procedural name abstraction.*

By abstracting out common, simple idioms, naming them then using them by name, the design discipline of building *modular procedures* supports the abstraction of simple first-order control and data idioms without resorting to fancy higher-order or generic procedures. Even low-level languages at least support this style of programming directly, if not the preceding two items.

Interestingly, the first two enumerated items above expand the domain of reach of the third by allowing programmers to abstract out common patterns in both control structures and data types, respectively, further enhancing the overall modularity of design. As argued earlier, by naming these control and data-type abstracted procedures and using them by name, programmers can modularize over high-order and generic entities.

There are probably a few more ideas one could include under the rubric of "high-level, abstract, modular programming", but these three suffice as a conceptual basis for the discussion that follows on the relative advantages and disadvantages of designing large, sophisticated software systems this way. The purpose of this subsection, after all, was merely to motivate what follows, not to exhaustively define a broad software design methodology.[2]

0.2.1 HAM is Desirable

As alluded to in the above explanatory snippets, employing high-level, abstract, modular programming offers several advantages over low-level, concrete, monolithic system design. Among them are:

a) HAM code suppresses detail through abstraction for rapid prototyping.
b) HAM code is expressive, allowing concise presentation of ideas in code.
c) HAM code is easy to read and understand due to its clarity of expression.
d) HAM code is easy to maintain due to its clarity and separation of ideas.
e) HAM code is easy to extend due to its deliberate methodical generality.

Again, there are probably other items one could add but these should suffice to illustrate the point. So-called *programming in the large* is well-nigh impossible without addressing each of these points and likely others as well.

[2] Specifically, it is not my intention here to have given short shrift to data-centric programming methodologies, such as in APL [Guibas & Wyatt 78], nor to discount purely functional approaches, such as that advocated in John Backus' FP system [Backus 78], nor to ignore any of a host of very high-level languages, such as those rooted in pattern matching and/or declarative programming, and so on.

To the contrary, many of the seeds for DESCARTES were sown in the fertile soil of dynamic compilation efforts as embodied by various APL compilers [Johnston 79] [Driscoll & Orth 86]. Seeing how APL compiler writers approached a much more general language gave me insight and clarity by contrast. My thanks specifically to Peter Szolovits for suggesting this line of investigation early on.

Moderation in all things; all things within reason.

Of course, this is not to suggest that every procedure in a large system be as high order and abstract and modular as possible. This would lead to code so abstract that it could mean anything and nothing.

0.2.2 But HAM can be Expensive

High-level, abstract, modular programming is not without drawbacks, however. Foremost among these is execution speed.

First make it work.
Last, make it fast.

Pervasive abstraction can slow down large systems. We will consider how shortly. But first, does speed really matter?

In many settings, execution speed is not critical. Often, in fact, its importance is superseded by functionality, extensibility, maintainability and attainability (*i.e.,* "time to market"). This is especially true in large, sophisticated, long-lived, production-quality productivity suites.

Speed doesn't matter.
Except when it does.
And then it's all *that matters.*

Unfortunately, however, when performance *is* an issue it tends to be a dominant issue: users may admire an elaborate software system's capabilities but if it is too sluggish, they tend to opt for less capable but more responsive alternatives. This has the unfortunate consequence of leading many software developers to decide early in the design process to opt for using programming languages and tools that are reputed to be "efficient" but whose long-term time-to-market and legacy maintenance costs may prove a poor trade off relative to the ease of development and downstream extensibility afforded by more flexible development alternatives. Prejudice is difficult to overcome in such cases without an acute appreciation of the *overall* cost of building and maintaining a large software system, integrated over the entire production lifetime. This is a particularly insidious instance of premature optimization, which many in the field consider to be "the root of all evil".[3]

In addition, even when time is not the precious resource being considered, the speed with which a program can execute a fixed workload also has implications for energy efficiency. Specifically, the fewer machine cycles needed by a machine to complete a task, the less heat will be generated and the less power will be consumed. These can be important considerations on mobile platforms where thermal throttling and limited battery life become restrictive. They can also be primary financial concerns when considering the cooling costs and electricity costs of operating large rooms full of server machines. In fact, ongoing operating cost scaling generally makes it more economical to buy and maintain a large number of slow, cool, low-wattage machines than to buy a smaller number of faster, hotter, higher wattage state-of-the-art speed demons. Performance is therefore indirectly implicated in economies of scale.

[3]This aphorism— "premature optimization is the root of all evil"— was popularized by Donald Knuth [Knuth 74] as a paraphrase of Sir Tony Hoare (according to The Fallacy of Premature Optimization by Randall Hyde [http://www.acm.org/ubiquity/views/v7i24_fallacy.html]).

In light of the above, one might rephrase the concern as one of execution efficiency rather than simply a matter of execution speed. Faster execution *per se* is not the objective: executing in fewer machine cycles is. Thus, performance per unit energy per unit time is the *true* optimization goal when speed is of the essence. Still, the overriding *global* goal should always be an end-to-end economical solution (including time, effort and expense in a wide host of areas such as development, maintenance, execution, power, capital expenditure, personnel, and so on).

The greater goal, therefore, might be better characterized as one of *high-productivity computing* in the end-to-end process, not merely *high-performance computing* of the final end product *per se*.

High-level, abstract, modular programming as a development methodology, as advocated here, can be a key component to this greater agenda of *high-productivity computing*. It is hoped, therefore, that the results of this dissertation will help to assuage some of the long-held prejudices against the HAM approach to software systems development.

0.2.3 Why is HAM Expensive?

This begs the question: what are the causes of this performance overhead in high-level, abstract, modular code?

Abstraction Leads to Pragmatic Over-Generalization

Cost may exceed value in small markets.

First, highly abstract programs can be excessively general in practice from a pragmatic run-time perspective.

For example, some higher-order procedures may be instantiated on only a few instances of some abstracted-out procedure argument or they may return a procedure that is only ever closed over a few instances of some abstracted out parameter. Similarly, some polymorphic procedures may never be invoked on more than a few different data types.[4]

In essence, this means that the extra level of indirection that abstraction introduces in the general case may impose an unfortunate run-time tax on those specific instances used in practice.

High-Level Code Leads to Interpretive Overhead

Second, high-level code can be difficult to compile directly to native machine instructions. Consequently, high-level languages often resort to an intermediate "virtual machine" layer essentially to byte-code interpret their more exotic constructs through various dispatch-driven mechanisms.

[4] Urs Hölzle [Hölzle 94], for example, found most polymorphic generic method instances in a representative suite of object-oriented programs to be used on only a small handful of concrete instances— typically around 5 but almost never more than 10— except for the occasional "*megamorphic*" procedure, such as fully general system utilities to print data to the output console and other core library facilities of that ilk.

Again, even in the non-OOPS setting, the ubiquitous quicksort (`qsort`) and binary search (`bsearch`) utilities of the POSIX library are classic concrete examples of higher-order general routines that tend to be instantiated on only a few prominent instances.

A classic example of this is the highly-general *Meta-Object Protocol* of the COMMON LISP *Object System* (the so-called CLOS MOP) [Kiczales, des Rivières and Bobrow 91]. This is an extremely flexible mechanism that allows the programmer to locally override nearly all aspects of the default generic procedure method dispatch and inheritance mechanisms, leveraging a wide spectrum of run-time and compiler-time customization criteria. Used unwisely, however, it can result in notoriously inefficient code, though not always obvious to the inexperienced MOP programmer.[5]

A LISP *programmer knows the value of everything, but the cost of nothing.*

— ALAN J. PERLIS (1922–1990), *Epigrams on Programming*, No. 55

The following paragraphs explore this vague complaint in specific detail.

Interpretive Mechanisms for Generic Procedures. Generic procedures, for instance, often incur run-time overhead to coerce their arguments into a canonical format before dispatching to a core non-generic instance-specific implementation.

An example is generic arithmetic that accommodates blending of integers, reals, rationals, complex numbers and so on. At run time, mixtures of arguments are coerced, say, to a least upper bound (most specific generalization) before being combined by base operators that are defined only on homogeneous argument lists for each distinct base type (integer, real, *etc.*). This run-time strategy of test-coerce-and-dispatch introduces interpretive overhead on code that might otherwise be streamlined if dynamically selectively specialized for the common cases that occur in practice.

Dynamic Redundancy in Interpretive Mechanisms. Moreover, this interpretive overhead is often dynamically redundant, as when several generic procedures are nested or cascaded within the body of some procedure. For example, consider this fully generic code fragment:

```
(minus (plus (times a (square x))
             (plus (times b x)
                   c))
       (times 4 (times a c)))
```

Here, each generic call out will dutifully test, coerce and dispatch on their arguments *each time they are called* then throw away this useful information immediately upon return. If each generic out-of-line procedure were instead in-lined, these redundant tests could be elided via standard *common subexpression elimination* (CSE), but since they are out-of-line calls, such localized situational optimization is not forthcoming. Therefore, requiring each generic operator above to independently test and coerce their arguments without the ability to communicate or

[5] Although invaluable as an exploration tool for advanced concepts in programming language development, the CLOS MOP's reputation for having unpredictable performance when used carelessly, even by seasoned COMMON LISP CLOS programmers, garnered the MOP a reputation for being a tenuous choice for performance-sensitive, production quality code, except when absolutely essential. This tended to relegate it to mainly academic interest. Still, to a greater or lesser extent, some of its functionality appear to have influenced contemporary object-oriented languages— such as JAVA and C$^\sharp$— under the guise of so-called "*reflection*" mechanisms and/or the rubric of "*aspect-oriented programming*". For details, see Éric Tanter's doctoral dissertation [Tanter 2004].

otherwise share their intermediates with one another is dynamically redundant. This is so because, once the first generic procedure determines that, say, some argument is an integer but needs to be coerced into a real, all subsequent temporally downstream sibling generics in the same lexical scope and dynamic extent within the procedure body should have direct access to these *desiderata*: access both to the fact that it is an integer *and* access to its coerced real counterpart and initial integer values.

Generic procedures should not harass their data.

```
(minus (plus (times a (square x))
             (plus (times b x)
                   c))
       (times 4 (times a c)))
```

Figure 0-0: A fully generic code fragment

In the above example, for instance, variables `a`, `x` and `c` appear more than once, but sibling nodes in the dynamic call graph cannot share intermediate coercions of, say, an integer `c` to floating point, should it be needed both to add it to the result of a floating point intermediate `(times b x)` and to multiply it by a floating point `a`. Because both `plus` and `times` drop their intermediate coercions of `c` upon return when called out of line, this coerced intermediate *desideratum* is dynamically re-computed by each. Thus, efficiency may scale poorly in the limit.

This I dub "*cross-tree redundancy*". It results from an inability to communicate, preserve and export desired intermediate values (such as coerced variables occurring as leaf nodes of the nested expression call graph) across sibling nodes of the process tree. One might imagine an exotic *simultaneous multi-set representation* of each datum so that coercions might be retained as sticky associated intermediates, but deciding when to do this and how long to preserve sticky intermediates merely compounds the problem. And yet this is effectively what is accomplished by selective in-lining and retaining of intermediates in machine registers, by virtue of CSE.

In addition, even producer/consumer redundancies can occur. For instance, in the above code snippet, the inner call to `square` may determine that `x` is a floating point value, and hence so too will be its squared result, and so on up the nested call chain. But, alas, the innermost `square` subexpression cannot directly communicate this to its "up-tree" ancestral caller nodes: `times`, `plus` and `minus`. Consequently, these consumers must redundantly test their in-coming first arguments to dynamically re-discover what `square` already figured out: *viz.,* they are floating point values. In essence, callees (like `square`) have no direct mechanism for reaching into their callers and short-circuiting forgone type tests on their returned results. Yet selective in-lining produces just this sort of knock-on effect, again through standard CSE.

This I dub "*up-tree redundancy*". Here the *desideratum* that the code fails to export across calls is not an intermediate concrete value *per se* but, rather, the consequent forgone conclusion of temporally downstream (lexically up-tree) latent value predications (like pervasive type tests). These in general could involve any sort of common predication used to steer computation, such as whether a value is odd/even, pre-sorted/non-sorted, dense/sparse, or prime/composite, *etc.*

To summarize, these export *desiderata* should be accessible at will to all siblings and ancestors *without* having to re-test and possibly re-coerce the same data repeatedly within the "*code neighborhood*" (*i.e.,* the siblings and ancestors of the lexical call graph and/or dynamic process tree) of a given procedure body's nested expression tree. As argued, selective in-lining facilitates this by virtue of standard *common subexpression elimination* (CSE) [Aho & Ullman 86].

Unfortunately, deciding where and to what extent to judiciously in-line these calls out to subordinate generic nested subexpressions merely redefines the problem without directly solving it. This is precisely where profile-driven program specialization comes into play.

The Core of the Interpretive Overhead Concern. To distill the preceding to its core, the objection that high-level code leads to interpretive overhead boils down to the following:

Information Flow: Cross-Tree Cooperation. As illustrated above, this is not the same idea as the popular trick of inter-procedural memo-ization or method caching across procedure calls: it is a problem of *intra-procedural sibling information flow* within a single procedure body and dynamic extent, not a problem of cross-procedural institutional memory over time or between textually decoupled caller and callee. Thus, this is a purely local matter (in both scope and extent) among subexpressions of manifestly or latently highly nested forms.

To review, for instance, once some incoming data are identified as integer or real and coerced appropriately by the first generic call out that manipulates them, all subsequent generic calls on them throughout the remainder of the procedure body might be streamlined accordingly by sharing these coerced intermediate values. This can be accomplished by replacing suitable generic operators by direct calls to their concrete base equivalents, passing the appropriate shared coerced intermediate values in place of their original argument counterparts.

Valuable pockets of operator specialization can result if only the constituent generic operators could somehow export these intermediate coercion *desiderata* directly to their siblings.

Information Flow: Up-Tree Percolation. In addition, knowing the ultimate coerced forms of arguments passed to a generic procedure generally uniquely determines the form of the returned result as well. Consequently, subsequent generic consumers of the intermediate results produced by nested generic expressions might all be dynamically collapsed into cascaded chains of direct calls to appropriate concrete base operators. In this way, the fruitful streamlinng of an entire nested expression tree can pivot on the dynamic type categorization of a single leaf element.

To review, for instance, real (floating point) numbers are contagious, so combining any kind of non-imaginary number with a real will generally produce a real result. "Once a `flonum`, always a `flonum`."[6] Therefore, generic ancestor operators invoked on real subexpressions will generally dispatch to their real concrete base operators, after coercing all their other arguments to reals. This portends a domino effect of potential dynamic specializations.

Valuable cascades of specialization can result if only the leaf expressions could somehow export these type categorization *desiderata* directly to their ancestors.

[6] Strictly speaking, the contagion is on the `inexact`ness attribute of `flonums`, not on the `flonum`ness attribute itself, but since there are no `exact flonums`, conflating the two is natural and commonplace in this narrow

The Core Information Flow Issue: Opaque Trampoline Interfaces. In effect, taken as a whole, the preceding is a complaint not only that generic procedures introduce a level of indirection between their callers and eventual low-level callees (in order to perform testing, coercion and dispatch), but that *they do so by interposing an* "opaque trampoline". *I.e.,* the entry point handlers for each generic procedure veil a test-coerce-and-dispatch "*trampoline*"[7] behind their statically opaque code module boundaries (a.k.a. their "*abstraction barriers*").

Specifically, if the trampoline code of each of several lexically nested (and therefore dynamically cascaded) generic siblings were open-coded into the caller, this pervasive in-lined test-coerce-and-dispatch code would be subject to standard local peephole optimizations— like *common subexpression elimination* (to avoid re-testing), *partial redundancy elimination* (to share intermediates), and *dead code elimination* (to manage in-lining bloat), just to name a few [Aho & Ullman 73] [Aho & Ullman 86]. Consequently, dynamically redundant testing would be eliminated, coercion counterparts would be shared across siblings, and the formerly opaque trampoline code would be streamlined in practice at run time.

Overhead due to Dynamic Call Graph Complexity

Finally, highly modular code can lead to excessively *deep* call chains when the many small modules that comprise a larger subsystem are deeply layered to inter-operate at run time to accomplish an intricate task. This can also lead to excessively *wide* call graphs when, say, each step of a multi-step sequence calls out to yet another small code module rather than executing monolithic native code sequences in line. Layering highly generic procedures, with wide multi-way genericity/specificity dispatchers, exacerbates both considerations.

These depth and breadth extremes in program call graphs, resulting from jumpy and branchy code, are the bane of code optimizers (referring both to the people striving for code efficiency and to the tools and mechanisms they build in its pursuit).

For instance, generic method calls and their attendant data-directed dispatch tend to result in a plethora of indirect branches/computed jumps. Such highly modular "hot potato" data

context. This conflation tacitly relies on one's familiarity with the standard containment relation among various classes of numbers, *viz.,* $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ and so on.

More broadly, this draws a distinction between the mathematical concept of integers, reals, rationals, and complex numbers, and their respective concrete machine representations as `fixnum`, `flonum`, `ratnum`, `slonum` (and `bignum` for large integers). The orthogonal consideration of `exact`ness is one of those peculiarities unique to SCHEME that one generally suppresses in forums such as this. It is of prized importance to the numerical analyst, but not germane to the more casual point being made here.

[7] A "*trampoline*" is an informal term to characterize small code snippets that provide an active level of indirection between a caller and its callee. Typically, they perform argument canonicalization, safety checks, variant dispatch, and so on, before proceeding to a target code block that can then safely presume the validity of the run-time code invariants established by said trampoline.

This is a more liberal use of the term than the definition given by the traditional computer science "jargon file" (published as The Hacker's Dictionary [Raymond 96]), which emphasizes that said code snippet be dynamically generated and often self-modifying. So, I use the term in the more general sense of, say, the (non-authoritative) entry for `"Trampoline_(computers)"` at the English Wikipedia [http://en.wikipedia.org].

When later I introduce on-the-fly, specializer-generated, run-time call dispatch stubs, these so-called "*poly-variant dispatch trampoline*"s (p. **??**) are more what a jargon traditionalist would call a "*true trampoline*". In that sense, I guess the more liberal usage here is an instance of a "*false trampoline*"?

and "pass the buck" control patterns often exhaust the patience (depth) and attention spans (breadth) of branch predictors, pre-fetch mechanisms and execution caches. Tragically, this can sabotage performance of these mechanisms at multiple execution layers: 1) at the language run-time level through branch biasing, execution caching and memo-ization/tabular-ization; 2) at the operating system level through dynamic working-set sensitive paging; or 3) at the hardware level both through microprocessor branch predictors and instruction pre-fetch units, as well as through memory system hardware caches.

Moreover, this quandary is not specific to object-oriented method-based programming nor generic procedure dispatch methodologies *per se*. For example, parameterized programming via code templates or compile-time macro expansion faces similar difficulties, as witnessed by the C++ *Standard Template Library* (or STL) [Stepanov & Lee 95]. Here, however, the added concern of "*code bloat*" arises, since compile-time template expansion is obligatory, not selective. Thus, HAM via macrology is no catholicon.

So the extra levels of call indirection introduced by pervasive modularity negatively impacts performance at all three major execution layers: the language run-time layer, the operating system layer and the hardware micro-architectural layer.

This is not a good thing.

Summary of HAM Expense

To summarize, HAM can be expensive because:

- High-Level code can incur interpretive overhead and dynamic redundancies.
- Abstract code can be overly general in practice, hiding instance detail.
- Modular code can exhaust dynamic optimizers due to jumpy and bushy call graphs.

Taken all together, these conspire to persuade many programmers instinctively to avoid HAM. One of the goals of this dissertation is to disabuse them of that predilection.

0.2.4 The Solution

Common among each of the above complaints is the observation that high-level, abstract, modular programming introduces levels of indirection in code that, at run time, carry a performance penalty. The obvious objective, therefore, is to selectively reduce this indirection overhead *without* having to manually re-write source programs in low-level, concrete, monolithic style. The natural solution is to do so automatically behind the scenes and to do so selectively, at run time, guided by dynamic profiling information indicating where it will do the most good.

Specifically, the dynamic overhead of higher-order procedures can be reduced by: a) selectively generating specialized versions of higher-order procedures for the most common instantiations of their abstracted-out data, and then: b) re-writing select call sites to call these specialized versions directly where appropriate and beneficial.

Next, specializing calls to generic procedures by selectively open-coding their test-coerce-and-dispatch trampolines based on profiles of dynamic data helps alleviate both the interpretive overhead of genericity and the redundancy it can impose. In fact, in the case where a specialized call to a generic procedure ultimately branches directly to a primitive procedure, that call can be open coded at the assembly language level via judicious compiler directives.

Last, selectively specializing and in-lining calls to general library and utility subroutines can flatten the modularity hierarchy, reducing the depth and breadth of the dynamic call graph. This reduces the pressure on dynamic execution optimizers like predictors, pre-fetchers and caches.

In each case, we selectively transform high-level, abstract, modular code into equivalent low-level, concrete, monolithic code. This is done automatically, transparently, safely, efficiently, simply and elegantly in a robust, rational, judicious, disciplined and economical manner (and whatever other coveted adjectives or adverbs you might want to throw in).

The body of this dissertation endeavors to support, in detail, most of these lofty boasts.

0.2.5 HAM Summary

To recapitulate, HAM code can accommodate many very general usage patterns that may never be exercised. The problem is that one cannot be sure which ones will and which ones won't be used until the code is actually run on some data set. Worse, these usage patterns can vary for different program users and they can change over time for a given user with varying input data sets reflecting ever-changing evolving objectives.

By using dynamic profiling information to choose where to aggressively specialize, in-line and open-code the high-level, abstract, modular procedures, we limit code bloat in a well-reasoned and disciplined manner. This results in effectively replacing performance critical code regions with streamlined equivalent code where it matters most, leaving non-critical code as is.

In short, selective program specialization reduces the wind resistance of high-level, abstract, modular code to make it streamlined and thus fast and efficient.

The remainder of this chapter gives a broad informal overview of how this is done.

0.3 Overview of the Descartes Prototype

The DESCARTES system marries a set of run-time profiling tools to a profile-driven kernel-level source-to-source program specializer and an optimizing compiler for the MIT C SCHEME dialect of LISP. This yields on-the-fly identification and specialization of performance-critical high-level, abstract program modules, selectively reducing them to streamlined equivalent code in a robust, disciplined, judicious and efficient manner.

0.3.0 Overview of Original Contributions

Enumeration of Original Contributions

The principal original contributions of this work (excluding tools), include:

1) *Spectral Specialization* - source-to-source program specialization with respect to *context spectra*, constituting dynamically adaptive, profile-driven, polyvariant on-line partial evaluation grounded in statistical inference; where...

2) *Context Spectra* - robust, compact and efficient representations of the point estimates of dynamic execution contexts to profile each procedure's dynamic data via weighted distributions over generalized structural types; with...

3) *Statistical Inference as Feedback* - the use of statistically inferred data as a rigorous feedback mechanism to focus, drive and throttle ongoing program optimization efforts while simultaneously regulating overhead through self-tuned parameters, thereby avoiding traditional architectural *ad hoc* limits (a.k.a. "magic constants").

These are fairly dense and technical points that may seem completely opaque without further explanation. Let's unravel them a bit to get a clearer idea of what's being claimed.

First, however, note from the outset that these *context spectra* do not encode just instruction-level branch prediction frequencies: they are an encoding of the source-level probability distributions on data, not just on instructions. This, for example, enables semantics-based program specialization that mere machine-level branch prediction frequencies in general do not allow.

Summary, Explanation and Elaboration of Original Contributions

Put simply, DESCARTES performs source-level code optimization by using dynamic profile information, all driven by statistical inference. The novelty lies in the nature of the optimization performed, what data is used to do it, and how the overhead of this ongoing process is managed. The original contributions listed above stake specific claims in each of these areas.

The following sections discuss each original contribution in turn.

Specialization over *dynamic data distributions* is the hallmark of Descartes.

Spectral Specialization is the Core Process. Specifically, *program specialization* is a well-established way to reduce high-level, abstract, modular code into lower-level, concrete, monolithic equivalent code. Normally, specializers optimize programs with respect to static values or types (or both). By contrast:

Descartes specializes programs with respect to *dynamic data distributions*.

This simple generalization is the key enabling insight for all that follows. It is the primary concept which guided the design of the DESCARTES prototype and which distinguishes this work from the previous advances in the field, on which it builds.

Making the simple complicated is commonplace;
Making the complicated simple, awesomely simple, that's creativity.

— CHARLES MINGUS (1922–1979)

Distribution-driven program specialization is the primary original contribution. It is the single most important idea of this work. Everything else is detail.

The details are not the details. They make the design.

— CHARLES EAMES (1907–1978)

Of course, how to do distribution-driven program specialization effectively and efficiently is the core engineering challenge this dissertation addresses.

The DESCARTES prototype demonstrates its feasibility through proof of concept, illustrating its viability with a few well-chosen examples while serving as a vehicle for exploring design trade offs and new ideas that arose from actually implementing this approach. It also serves as a fruitful vantage point for speculating on future extensions to this approach to program optimization.

Before elaborating on just how this is done, however, it is helpful first to examine the central data structure that drives this process: the distribution. This also happens to be the second original contribution of this work. We'll return to this first, primary contribution in due course.

The representation of *dynamic data distributions* is the cornerstone of Descartes.

Context Spectra are the Core Data Abstraction. So what are *dynamic data distributions*, how are they encoded, what do they look like, where do they come from, how are they used, and how much do they cost to produce?

What distributions are. Distributions are represented by *context spectra* in the DESCARTES prototype. Each procedure has its own associated *context spectrum*.

For instance, a procedure's *context spectrum* might indicate that its first argument had value 1 in 80% of its observed invocations so far and was a positive integer greater than 2 in the remaining 20%, while its second argument was always 0, and so on for each formal parameter and free variable appearing in the procedure's body (including those that occasionally might be unbound).

In short, *context spectra* capture precisely the kind of dynamic usage information needed to effectively optimize a program based on statistical expectations about the data that drive the execution process.

How distributions are encoded. Let's examine *context spectra* more closely. As these are layered data structures, let's do so by building them up from base components, standardizing terminology as we proceed. To that end, the progression of high-level ideas to carry away from all this goes from *dynamic execution contexts* to *dynamic abstract contexts* to *context spectra*—or DECs to DACs to *spectra*, if you like rampant acronyms and spurious terseness.

First, we define the *dynamic data set* of a procedure to be the set of formal parameters and free (non-local) variables that appear in the procedure's body. This set of names identifies the data that might influence the behavior of the procedure at run time but whose values, in general, are not statically known. Note that this definition intentionally ignores those formal parameters that appear nowhere in the procedure body: they are semantically irrelevant. More importantly, note that this definition is a purely static syntactic property of a procedure's source text: it is just a set of names dictated by the lexical scoping rules of the source language. I coin this term partly to avoid the wordier phrase "referenced formal parameters and free (non-local) variables" but also to contrast it with other interesting properties of the elements of this set besides just their names.[8]

Next, we define the *dynamic data sample* of a procedure at time t to be the set of dynamic values associated with its *dynamic data set* as they actually occurred at run time at the specified time t during the program's execution. Note that this definition of *dynamic data sample* is not quite the same as the familiar "procedure activation frame" since standard activation frames typically do not include the values of the procedure's pertinent global free variables whereas the *dynamic data sample* defined here purposefully does.

Now we can define the *dynamic execution context* (DEC) of a procedure at time t to be its *dynamic data set* paired point-wise with its *dynamic data sample* at time t. This defines the full dynamic local state of a given procedure at a given time. Conceptually, it captures a self-contained snapshot of the in-flight bindings needed to "close" the procedure's body so that it could in principle be fully evaluated/executed later by, say, a hypothetical off-line context-based interpreter as if the procedure had been invoked in the equivalent actual run-time setting.

Of course, retaining potentially large sets of run-time data, even just copies of them, even temporarily, could prove prohibitively expensive in practice and could incur substantial "probe effects" on the running program (by bloating the heap or thrashing the data cache and such, for example). Therefore, DESCARTES will generally never have occasion to actually capture a full

[8] Those conversant in electronic chip design might be tempted to call these simply "inputs", but that term typically includes the "unused inputs" that I explicitly seek to exclude.

dynamic execution context directly. Consequently, no such hypothetical off-line context-based interpreter has been implemented.

Instead, DESCARTES examines the *dynamic execution context in situ* as an ephemera to abstract out approximate value/type/shape information from its constituent *dynamic data sample.* One might think of it as the flash of a camera: it is there only as a phenomenon, not as a manifest physical object, and only for an instant, just long enough to record the image of the subject, not the actual subjects themselves.

We therefore define the *dynamic data assay* of a procedure at time t to be this abstracted out approximate value/type/shape information of its *dynamic data sample* at time t. If the sample is the subject of our snapshot photography then the assay is its image recorded on film. Note that I use the term "assay" here advisedly since some dynamic data may be unbound depending either on the presence of optional formal parameters or on the fluid binding state of branch-specific free variables. I'll ignore such bugaboos in the future since they can be detected and their "values" recorded as distinct "unknown" sentinels (spec., $\perp$). They are not needed to determine how a procedure's body might be specialized with respect to a given observed instance of its *dynamic data set* anyway. I mention them here only in order to dispense with them once and for all. Good riddance. Note also that I've been careful not to specify yet just what approximate value/type/shape information is actually abstracted out. That's because it is dynamically configurable and will vary over time, the details of which will become vital when considering how the program specializer uses assays but which is not vital for the present discussion. For now, imagine them as analogous to thermal images or contour maps or silhouettes or chiaroscuro, if that helps.

As you may have anticipated, next we define the *dynamic abstract context* (DAC) of a procedure at time t to be its *dynamic data set* paired point-wise with its *dynamic data assay* at time t. Whereas the *dynamic execution context* associates variables with concrete values, this *dynamic abstract context* associates variables with the abstract approximations of those concrete execution-time values. So, although the *dynamic execution context* (DEC) is useful as a chimeric concept, the DESCARTES prototype in actuality records *approximations* of DECs, not full DECs. These approximate DECs we call *dynamic abstract contexts* (DACs). If DECs are moments in time, then DACs are those moments captured on film, which is to say representationally, abstractly and, these days, digitally.

Finally, these *dynamic abstract contexts* are accumulated over time to build an abstract history of how a procedure has been used in the past. Each history is compressed incrementally on the fly by merging its new in-coming constituents into a point-wise histogram, an example of which we shall see presently. These we call *dynamic context spectra*, or just *context spectra* since a "static" context spectrum would be a histogram of only a single component, the static (and therefore unchanging) abstract context, which isn't so much a spectrum proper as it is a singularity.

Therefore, at last, we define the *context spectrum* of a procedure to be the histogram of its *dynamic abstract contexts* accumulated over some time interval by merging them into weighted point estimates of their constituent *dynamic data assays* collected over that interval. Thus, each procedure is associated with its own *context spectrum*, which encodes that procedure's accumulated weighted expectations for the various abstract assays of its dynamic data samples witnessed over some time interval.

In sum, *context spectra* encode the dynamic data usage patterns of procedures.

What distributions are *not*. In passing, one should note that these *context spectra* are distinct from the *call cache* of *call context*s generated in static control-flow analysis [Shivers 91].

In static control-flow analysis, *call context*s form a statically-inferred mapping from procedure call sites to their environment bindings ("data context"). In DESCARTES, by contrast, the *context spectra* are not the result of static analysis: they are directly measured from the run-time system. Therefore, whereas static control-flow analysis will be forced to provide conservative estimates of what procedures may be called from where, and what their parameters and free variables might be, DESCARTES instead accumulates a *dynamic data distribution* to quantify this information, complete with statistical relevance and confidence bounds.

The result is that DESCARTES has more accurate information about procedure behavior as well as error bounds on the quality of that information. As this behavior changes over time, mirroring either subtle or substantial changes in program usage patterns, the *context spectra* will reflect these emergent phenomena.

This dynamically adaptive workload characterization and behavioral tracking is beyond the purview of purely static techniques. It is one of the hallmarks of dynamic profiling.

Moreover, the primary intent of mainstream static control-flow analysis using *call context*s is to identify (inductively) which procedures are called from what call sites. The goal there is to perform *closure analysis* so that one can statically determine an estimate of the dynamic call graph. The purpose is to recognize constrained instances of otherwise unbridled recursive procedures in order to reduce this statically estimated call graph to simplest form.[9]

[9] Specifically, the ultimate aim is to avoid having to use a restrictive special syntactic form (*e.g.,* `letrec` or local `defines`) in order to support recursive procedure definitions efficiently. Instead, using free-form higher-order recursive operators to attain the same result is preferred, since this is more general and carries less syntactic baggage for the language implementor, for example.

The classic instance of such a recursive operator is Curry's famous *minimal fixed point operator* (a.k.a. Curry's "*paradoxical combinator*") [Curry 42] [Curry & Feys 58/68] [Stoy 77], which in call-by-value semantics is:

```
(define Y (lambda (h) ((lambda (w) (w w))
                       (lambda (x) (h (lambda () (x x)))))))
```

Using this, one can define the Fibonacci series as the least fixed point of a tree-recursive functional, such as:

```
(define F (lambda (g) (lambda (z)
                        (if (< z 2)
                            z
                            (+ ((g) (- z 1))
                               ((g) (- z 2)))))))
(define fib-x/Y (Y F))
```

Fixed point aficionados will note that this works because `(Y F)` ≡ `(F (Y F))`. Better compiled code is produced for `fix-x/Y` when it can be determined that `g` is only ever bound to `(lambda () (x x))`. Such conundrums are a central concern for traditional static control-flow analysis [Rozas 84] [Shivers 88] [Harrison 89] [Shivers 91] [Sestoft 91] [Rozas 92] [Ashley & Consel 94] [Might & Shivers 2006] [Might 2007] [Might & Shivers 2007].

In DESCARTES, profile distributions capture more accurately (and significantly more easily) this dynamic call graph and *dynamic data distribution* information that these more complex static approaches seek via fixed point induction. DESCARTES simply observes them directly.

Specifically, if periodic procedure sampling reveals a given closure's compiled code body (its "*closure sanctum*") to be active sufficiently often, a data breakpoint is set to accumulate the *dynamic data distribution* of its formal parameters and free variables. This combination of a procedure's statistical relevance and dynamic context information supplants static *call context* approximations to the extent that the profiled code and its *context spectrum* constitute a (statistically weighted) mapping between closed-over procedure bodies ("*closure sancta*") and their enclosing environment bindings ("*call context*"). This is done without the need to inductively construct an approximate dynamic call graph, since dynamic context sampling records the dynamic call chain directly as part of the *context spectrum*.[10]

The availability of this statically opaque usage pattern data in a quantifiable and error-bounded encoding is the cornerstone of the dynamic approach. It is what drives the specialization enterprise and focuses its efforts.

This is not to say, however, that static methods are to be shunned. To the contrary, when static analysis *can* prove exact results, it can use that information to great benefit. The two approaches are therefore complementary. In fact, DESCARTES exploits this by using the stock MIT SCHEME compiler ("LIAR") to compile its generated specialized code. LIAR does just this sort of static *closure analysis*, among other fancy tricks [Rozas 84] [Rozas 92]. In this way, DESCARTES leverages the advantages of both approaches in tandem to produce optimized code that neither alone could attain.

This dynamic approach complements and augments static methods. It does not, in general, subsume them.

What distributions look like. Returning now to the topic at hand, the low-level details of the encoding of *context spectra* will become important when we examine closely how they are used to generate program specializations but, for now, broad strokes suffice. Let's examine a simple example to build our intuition before returning to how they are used to specialize programs.

A simple example. Consider, for example, the following simple procedure, `fib-x/no-ui`, and a *context spectrum* taken from a typical data profiling run.

[10] To wit, the DESCARTES breakpoint data profiler tracks the *dynamic link* (return/environment continuation) of the execution stack in order to parse the *dynamic execution context* while profiling dynamic variable bindings (the transitive closure of which constitutes the formal parameter and free variable bindings of the in-flight closure). The complete string of such *dynamic link*s live on the stack constitutes the full ancestral dynamic call chain of the procedure invocation being profiled. This is accumulated as part of the *context spectrum*, as the first slot of each context frame records the *dynamic link* (return continuation) that gave rise to it (as we shall see shortly). Thus contextual frames are call-site specific.

```
;;      ----- -----
(define (fib-x/no-ui z) ;;; ''no-ui'' means no open coding/early binding of primitives like '+', '<', '-', . . .
  "
  ----------------------------------------
   Fibonacci -- exponential algorithm
  ----------------------------------------
   Cheat: identity on z < 0; fib on z > 0
  "
  (if (< z 2)
      z
      (+ (fib-x/no-ui (-1+ z))
         (fib-x/no-ui (-   z 2)))))
```

This computes the $\mathtt{z}^{\mathrm{th}}$ element of the Fibonacci series using the ubiquitous tree-recursive (exponential) algorithm without resorting to the "usual integrations" (like automatic open-coding of primitive operators), hence its peculiar name.

When invoked with argument 39, its *context spectrum* looks like this:

```
(#(#(#[compiled-procedure 33 (fib-x/no-ui "fibs-no-ui" #x1) #x14 #x145D0F8] -->
      #[lambda 36]                                                          ==>
      (fib-x/no-ui z))
    #(z
      =
      #(hetero-type
        #(204668309
          ((exact-positive-one     30.9017 . 63245986)
           ( tiny-positive-fixnum  30.9017 . 63245985)
           (exact-positive-two     19.0983 . 39088169)
           (exact-zero             19.0983 . 39088169)))
        ()
        ())))
  .
  204668309)
```

This unsightly mess is parsed as follows.

First, the *context spectrum* shows that `fib-x/no-ui` is a compiled procedure (object 33) appearing as the first object defined in a file named `"fibs-no-ui.scm"` and its entry point resides at byte offset 20 (`14h`) at hexadecimal address `145D0F8` in the run-time heap. Its source code corresponds to `LAMBDA` object 36, which is a procedure of one argument, `z`. The procedure itself is named `fib-x/no-ui`.

During this profile run, `fib-x/no-ui` was invoked some 204,668,309 times. Of that, its argument `z` was observed to have a heterogeneous variety of values. To wit, some 30.9017% of the observations (spec., 63,245,986 of the 204,668,309) were of the constant 1. One fewer observations (63,245,985) were to a positive integer greater than 2 but less than a "tiny"-ness threshold. The remaining observations were evenly split between the constants 2 and 0. In all cases, `z` was an atomic datum. Never was it an aggregate data structure, neither of fixed length nor of variable length. (For example, it was never a linked list nor a vector.)

By the way, this was a *complete* profile run of Fib[39]. Every single recursive call was halted, its *dynamic abstract context* was ascertained and merged into the accumulated *context spectrum*, then computation was resumed. This was done during the entire Fib[39] run, start to finish.

Note, for example, that the ratio of instances of two adjacent elements in the series indeed approximates the golden ratio, ϕ:

```
(/ 63245986.0 39088169.0) ⟹ 1.6180339887498951
```

This is accurate to 14 decimal places (since the last two digits should be 48, not 51). This observation is reassuring, if mathematically obsessive.

This was also extreme overkill. One doesn't need 14 decimal places of precision to determine that this procedure would benefit from being specialized for argument 1 and for tiny positive integers since around one third of the time it is called on those kinds of arguments. Only around one fifth of the time is it called on 2 or 0 so those are perhaps less important to generate additional specializations for if there are more vital specializations pending for other procedures.

Ballpark relative proportions suffice.

What our simple example reveals. I belabored this example somewhat to illustrate a few points about what *context spectra* contain:

i. [Procedure]: Information about the procedure profiled— where its executable code resides, where its associated source code is, and what its formal parameter list looks like— are all recorded. This example contained no free variables other than primitive procedures in operator position, which are treated specially, so we ignore those for now.

ii. [Data]: For each formal parameter and free variable, a sorted point estimate histogram of its abstract value distribution is accumulated. In this case, only z's scalar data distribution was observed.

iii. [Stats]: Taking complete profiles by interrupting every call to a target procedure is excessive. All we are really interested in are the relative proportions of the spectral instances of the *dynamic data set* up to some reasonable level of statistical confidence. Total population descriptive statistics, therefore, appear to be information overload.

Note that embedding the relative frequency of each distinct constituent type for each formal parameter can guide the decision of how aggressively to optimize for each distinct case based on statistical expectation. Moreover, by also embedding the raw instance counts of each observation, one can compute the confidence intervals for each observed type expectation based on sample size.

For example, given the proportion of instances of 1 for z and the total number of observations, one computes the confidence intervals for the standard 95% and 99% bands as follows:

```
------------------------------------------------------------------------
(* 1.96 (sqrt (/ (* 0.309017 (- 1 0.309017)) 204668309)))  ;;; 95% level
;Value: 6.33075691484453e-5
------------------------------------------------------------------------
(* 2.58 (sqrt (/ (* 0.309017 (- 1 0.309017)) 204668309)))  ;;; 99% level
;Value: 8.33334328586678e-5
------------------------------------------------------------------------
```

Therefore, assuming the underlying distribution in the above profile run to be representative, the 95% confident band for the parameter `z` being observed to have value 1 spans the interval 30.9017%±0.0063%. This means we can be 95% confident it will have value 1 somewhere between 30.9080% and 30.8954% of the time in the future. Even at the 98% confidence level, the same result obtains (to four decimal places, anyway).

Collectively, this data strongly implies that profiling this procedure on every single recursive call is statistically excessive even for small inputs like 39. It would be as effective, and far more efficient, to profile only a subset of these calls during the program run. This argues for doing statistical sampling and using statistical inference to measure confidence rather than doing a complete survey and using descriptive statistics.

We'll revisit this same observation from a different angle when considering the run-time overhead of profiling. (See below, § 0.3.0, p. 29.) Meanwhile, suffice it to say that sampling is our friend. Replacing descriptive statistics over complete populations with statistical inference over sample distributions makes this transition painless: in fact natural and easy.

Moreover, one could well argue that, at any given moment during a program's lifetime, any profile that has been accumulated is already, in effect, merely a partial sample of the conceptually infinite population extending into the future. Under this view, a statistician might refer to the current profile of the moment as an *opportunity sample*, meaning it samples the first N instances that opportunity presents. If at some point we decide we have sufficient data (like, our confidence interval has narrowed below some threshold), then sampling can cease, or at least be suspended temporarily. Conveniently, this motivates our next consideration.

Where distributions come from. These distributions are generated dynamically via run-time execution profiling.

They change over time as procedures are invoked from various call sites and on varying input data sets. By doing so, they track the weighted distributions of each procedure's abstract *dynamic execution context* as they evolve. For each observed *dynamic data sample* of a procedure, for example, the configuration's broad-spectrum value/type/shape information is abstracted out, encoded, and merged with that procedure's accumulated *context spectrum*.

This is done in the DESCARTES prototype through compiled code breakpoints, a standard technique whereby a procedure is interrupted upon entry, some computation is performed within the *dynamic execution context* of the interrupted procedure, then execution proceeds as normal. The computation performed at the breakpoint is done by a *breakpoint handler*, a general procedure that takes as argument a pointer to the interrupted procedure, a pointer to its run-time stack, and a special return ticket used to proceed the interrupted execution. The breakpoint handler does not return a value: it is called solely for effect— namely, it parses the *dynamic execution context*, abstracts from that a *dynamic abstract context*, and merges that into the accumulated *context spectrum* for the breakpointed procedure.

This illustrates that nothing particularly magical is required to produce dynamic *context spectra* from running programs. Standard methods suffice.

How distributions are used. The *context spectrum* of each procedure reflects the accumulated profiles of presumably many invocations from potentially many characteristically distinct call sites. That is, a procedure may be used on characteristically different data sets from textually distinct call sites or even from the same call site while operating on multiple data sets.

Consequently, any given procedure might be specialized multiple ways into potentially many variants if its distribution of distinct execution contexts suggests such effort is warranted. This is done using a generalization of a standard technique called *polyvariant on-line partial evaluation*.

In short, *polyvariant on-line partial evaluation* means that multiple specializations (polyvariant) are generated for the many call sites of a procedure [Ruf 93]. The specializations are generated by a special kind of abstract interpreter that walks over the program source, along with the abstract information about its parameters and free variables (the *spectral context*), rewriting the source code by performing as much evaluation and specialization as possible while leaving unaltered those subexpressions where no such semantic reduction is possible. This particular kind or abstract interpretation, where source code is transformed to simpler equivalent source code, is called *partial evaluation*.

Doing so by carrying the abstract input data along to direct the specialization efforts makes it (one-phase) *on-line partial evaluation*. It is "on-line" in the traditional sense of being performed "as the program runs", the program in this case being the specializer. By contrast, (two-phase) *off-line partial evaluation* makes a pre-pass to annotate which expressions to reduce/simplify followed by an evaluation pass to actually perform the reductions.[11]

One could also call this *polyvariant specialization* but, to me at least, that terminology emphasizes the objective more than the method by which it is achieved: *polyvariant specialization* is the goal; *polyvariant on-line partial evaluation* (over *context spectra*) is the process by which that goal is attained. Still, I'll use the two somewhat interchangeably depending on what is being emphasized, the goal or the process.

By performing *polyvariant specialization*, potentially many distinct streamlined versions of performance-critical procedures are generated based on their various and evolving patterns of usage at various call sites over time. In the limit, this could involve overwhelming memory with a potentially unbounded number of specializations of even a single highly-general procedure. Worse, if aggressive optimizations like in-lining/unfolding are performed along cyclic call chains

[11] In passing, it is this firmly established use of "on-line" vs. "off-line" within the partial evaluation community that prompts me to characterize profiling and specialization as the application program runs as being "dynamic" vs. the traditional "static" approach of specializing code at compile time based solely on compile-time available information. Otherwise, to call DESCARTES "on-line *polyvariant on-line partial evaluation*" would be confusing at best, if not grammatically offensive. « *C'est la vie.* »

(*e.g.,* through transitively recursive procedure definitions), this partial evaluation process might not even terminate.

Normally such concerns over code bloat and divergence are handled by a conservative limiting mechanism called *binding time analysis* (*BTA*) [Jones & Schmidt 80] [Mogensen 89]. Roughly speaking, this scouts out the terrain in advance, using static program analysis to anticipate and avoid these traps.

Simple versions of BTA simply arbitrarily restrict in-lining and unfolding to fixed bounds. This, in effect, imposes an *ad hoc* bounded time and space threshold on the partial evaluation process. Less simplistic BTA mechanisms employ sophisticated methods such as Kleene fixed point induction [Kleene 50, § 66, Theorem XXVI] [Cousot & Cousot 77] and/or static call graph inference/control-flow analysis [Rozas 84] [Shivers 91] [Sestoft 91] [Rozas 92] [Ruf 93] to achieve the same effect while avoiding *ad hoc* threshold limits.

These more advanced techniques can be computationally expensive, however, if not done carefully [Ashley & Consel 94] [Christensen, Glück & Laursen 2000]

Worse, BTA based solely on static program analysis can still encourage over-optimization of seldom-executed program paths while neglecting the common path. This results in wasted effort in the former case and missed opportunity in the latter. Programmer annotations can be used to direct the effort but programmers are notoriously unreliable at predicting the performance-critical paths of their own code [Bentley 82]. Moreover, even when programmers get it right, cluttering the code with *deus ex machina* pragmatic declarations creates maintenance friction by introducing non-obvious directives that must be kept up to date as the software system evolves.

Of course, BTA adherents argue that off-line profiling can be used to automatically generate these annotations based on characteristic test suites, but this just shifts the problem to one of producing a representative characteristic test suite that will suit all users over all time. Either that or one must produce a collection of such test suites and rely upon the user to select the one among them that perhaps best matches how they think they use the code.

As stated earlier, a better way to address this is to continually profile how a given program user uses a program over time, using this dynamic expectation information to direct the specialization efforts. This dynamic distribution information can change over time, adapting to the ever-changing usage patterns of a user whose goals and data sets evolve. Moreover, this information can be passed directly to an on-line partial evaluator without requiring an off-line pre-pass to embed the information as source code annotations.

This is the approach taken in the DESCARTES prototype. It is the heart of the thesis: that dynamically adaptive specialization is desirable to effectively improve program efficiency and that it is efficiently attainable in practice.

Specifically, by building up *context spectra* to embed point estimates of the relative proportions with which pertinent dynamic data attain particular relevant characteristics, these spectra can be used as the abstract input information to drive the specialization process directly. This focuses the specialization efforts along those code paths that are relevant in practice, ignoring those that are not critical. Further, by replacing strategic call sites with calls to specialized versions of procedures where appropriate, the effective dynamic call paths are reduced in depth and breadth along those paths that most impact the program's run-time behavior. All this is achieved through a modestly generalized form of a standard optimization method: *polyvariant on-line partial evaluation* [Bulyonkov 88] [Ruf 93] [Malmkjær & Ørbæk 95].

This modest extension involves simply having the partial evaluator use *context spectra* rather than the usual traditional non-statistical static value/type information as input. This empowers DESCARTES to use the embedded relative statistical weights to drive how aggressively to pursue in-lining and/or unfolding and to steer which code paths to pursue, throttling to a statistically reasonable depth. All this is accomplished in a robust, rational, judicious, disciplined and economical way by virtue of being grounded in statistical inference.

How much distributions cost to produce. To this point, we've seen suggestive fragments of how DESCARTES can be effective at improving program efficiency. We have yet to consider, however, how costly it might be to produce these *context spectra* in the first place. Let's address that now.

As sketched above, *context spectra* are produced by compiled code breakpoint handlers that parse the *dynamic execution context* to abstract out a *dynamic abstract context* that is then merged into the accumulated *context spectrum*. This raises concerns of how expensive it is to parse, abstract and merge context data.

This is no idle concern. For example, consider again the Fibonacci example:

```
;;      ----- -----
(define (fib-x/no-ui z)
  (if (< z 2)
      z
      (+ (fib-x/no-ui (-1+ z))
         (fib-x/no-ui (-   z 2)))))
```

A simpler non-trivial procedure can scarcely be imagined. So what is the overhead in breakpoint profiling its *context spectrum*? Let's simplistically pretend that each primitive procedure takes a single machine cycle to execute. Each recursive call, therefore, takes 1 cycle for the `<` comparison then, in the non-base case, an additional 3 cycles for the `+`, `-1+` and `-` operations, plus the cost of 2 recursive calls out to `fib`. We should also count 1 cycle to look up each identifier referenced, both parameter names and operator names, primitive or otherwise, roughly doubling the above counts. Precision is not the goal of this informal "back of the envelope" calculation, so let's just pretend each count above is doubled.

We could define and solve a recurrence relation using these per-loop constants to roughly predict the total number of machine cycles needed to compute the $\mathtt{z}^{\text{th}}$ Fibonacci element, but it suffices for our purposes merely to consider only how the per-loop constants change due to breakpoint profiling. In the above simplistic analysis, we get something like a total of $2x1$ cycles per call in the base case and $2x(1 + 3 + 2) = 12$ in the induction step.

Now consider how many cycles are needed for the breakpoint handler to parse, abstract and merge the *dynamic data context* each recursive call. One can easily imagine straightforward methods to accomplish each of these three tasks taking only 1 cycle each per parameter or free variable to find the datum within the *dynamic execution context*, to abstract out its value/type/shape, and to merge the result into the accumulated spectrum.

For the Fibonacci example, this amounts to $3x6 = 18$ additional cycles, 3 per parameter (*viz.,* `z`) and free variable (the four primitive operators plus the recursive reference to Fib itself, since that could change each loop through self-modifying code). Thus, our original 12 cycles per loop has grown by 18 additional cycles to 30 cycles. This is a factor of 2.5.

Although this will be less substantial for larger code blocks (since the breakpoint overhead grows with the number of formals and free variables, not with the code block size *per se*), it is unavoidable that the breakpointing overhead will be relatively high for small code blocks. In a system that exploits modular software design and makes liberal use of small library routines, small code blocks will be the expected norm. Moreover, in typical programs, it is generally true that much, if not most, of the time is spent in library routines. This is a problem.

Worse, the above discussion naïvely presumes that dynamic values are always treated as scalar, atomic data. Knowing only that some datum is a linked list or vector, however, is not as useful as knowing more precisely that it is a list of integers, or a vector of reals, or a vector of pairs of lists of ... *ad nauseum*. This suggests a new, more dire concern: that parsing, abstracting and merging shape (structural type) information for aggregate data (like lists and vectors) grows with the size of the data, not simply with the number of parameters and free variables. Even modestly simple situations can be worrisome in light of this observation, not to mention the tribulations of cyclic data.

For example, consider the following simple yet less trivial example program:

```
(define (sc-* thunks)
  "-----------------------------------------------------------------------------
   (SC-* thunks) -- short-circuit product of thunks called left-to-right.

    If any thunk call yields 0, short circuits to return 0 w/o calling any of
    the thunks 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)))
```

Just what this program does is not important here. We will revisit it in more detail in the coming chapters. It is part of the Bayesian belief network case study.

What is important is what the *context spectrum* of the internal `DO` iterator looks like (Figure 0-1, p. 28).

Of particular interest here is the `thunks-to-do` parameter of the internal `DO` loop. Some 20% of the time it was a null list (an atomic type) while the remaining 80% of the time it was a linked list of various lengths. Of the 80% of the time it was a list, some 28% of that time it was of length 1 and 28% of the time a 2-element list, the remainder of the time being evenly split between being a list of length 3 or 4. In all cases, regardless of length, these lists were composed of compiled entry points (compiled procedures), namely, the *thunks*[12] [Ingerman *et al.* 61].

[12] A *thunk* is a procedure of no arguments, corresponding to a niladic functional closure. The term originated

```
(#(#(#[compiled-return-address 42 ("proposal-code" #xA) #x40 #x145FAE4] -->
     #[lambda 61]                                                     ==>
     (named-lambda (do-loop thunks-2-do acc)))
   #(thunks-2-do
     =
     #(hetero-type
       #(19.6446 null 22110)
       ()
       #(80.3554
         (list-type
          (((1 27.76426 . 25110)
            (2 27.76426 . 25110)
            (3 22.23574 . 20110)
            (4 22.23574 . 20110))
           .
           #(compiled-entry compiled-entry compiled-entry compiled-entry)))
         90440)))
   #(acc
     =
     #(hetero-type
       #(112550
         ((        positive-flonum  73.24745 . 82440)
          (  exact-positive-one     22.31008 . 25110)
          (  exact-zero              2.66548 .  3000)
          (inexact-positive-one      1.77699 .  2000)))
       ()
       ())))
 .
 112550)
```

Figure 0-1: Pedigree – Short-Circuit Multiply – Context Spectrum

In general, therefore, for arbitrary sizes and shapes of aggregate data arguments, each of the three phases of context profiling (*viz.*, parsing, abstracting, and merging) could take arbitrary time to perform, with comparable *space* required to accumulate the abstracted data. Space concerns can be mitigated through representational canonicalization (hashing to equivalence class representatives, for example) but the time overhead is not so easily dispatched.

How then can we regulate the overhead of context profiling in the face of unpredictable per-breakpoint overhead? The answer once again is to leverage the magical tool that is statistics: statistical sampling to the rescue!

To review what was noted earlier (§ 0.3.0, p. 21), *complete* profiles already have been shown to be overkill from the standpoint of information overload. Only relative proportions of occurrences of distinct value/type/shape instances are useful, not complete measures. Statistical sampling was offered as a way of exploiting this to manage excessive information. Specifically, the standard *opportunity sample* was mentioned as an obvious and familiar example sample selection mechanism in order to make a natural transition from complete to partial profiles. Simple.

Likewise, statistical sampling also can be used to manage the overhead of the data profiling process. It is straightforward, for example, for a system to keep track of what time is spent in the profiling code versus time spent in the process being profiled. When the ratio of the two becomes excessive by some metric, the profiling process can be throttled until the ratio improves. For intermittent opportunity sampling, for example, this would mean lengthening the delay between taking successive *opportunity samples*. In essence, this allows the system to amortize the overhead of profiling by spreading it out over larger time intervals to reduce the average overhead in the limit.

DESCARTES is instrumented to use just such a self-tuning feedback mechanism, but not by tuning delays in successive *opportunity samples*. The next main original contribution addresses this point.

Statistical inference is the unifying principle that makes Descartes viable.

Statistical Inference as Feedback: The Tie that Binds. In the preceding original contribution point, we considered statistical profiling as an alternative to complete profiling as a way of managing both excessive profiling data and excessive profiling time. We also considered how the time spent profiling can be managed by dynamically adapting the interval between successive profiling sessions to limit the overhead to some target percentage.

This trades overhead against responsiveness, governed by a target confidence limit in the measured variance estimates that are embedded within constituent elements of *context spectra*. Specifically, for a fixed target confidence limit, the number of sample data points required to achieve that level of confidence in the collected profile is essentially constant. Varying the time interval between sample events in order to lower the relative profiling overhead merely makes

in connection with ALGOL 60 [Ingerman *et al.* 61] and thrives in the functional language community (*e.g.*, [Hatcliff & Danvy 96/97]).

the profiler take longer to reach the same goal. This, therefore, makes the overall adaptive system a bit less responsive to change, especially short bursts of anomalous behavior.

In this section, I further develop this idea of amortized dynamic optimization overhead through statistical, ratio-driven, self-tuning feedback as an important original contribution in its own right.

The core idea of statistics-driven self-tuning feedback control will be familiar to students of operations research or financial investment or other related disciplines. Fundamentally, it is an issue of controlling investment effort based on running averages and expected cost/benefit.

The third original contribution of this work lies in applying analogous statistical inference techniques to both data collection and process control, at all levels of a self-monitoring, self-tuning, multi-level system of cooperating sub-processes. This trades profiling overhead against the system's adaptation responsiveness in a mathematically disciplined manner governed by a (dynamically variable) target confidence limit.

To wit, DESCARTES employs this idea at three distinct levels:

0) ongoing run-time sampling of the virtual machine to discover bottlenecks and quantify their relative importance while monitoring the bottleneck discovery process to limit its overhead through self tuning (as we'll discuss next);

1) ongoing run-time profiling of these discovered bottleneck programs to produce ever-changing distribution profiles of program usage patterns at modest, self-tuning targeted cost; and

2) ongoing optimization of programs using both data sets to drive the optimization effort while simultaneously monitoring the relative run-time overhead of the optimization process itself to manage that dynamically in an adaptive self-tuning manner as well.

What makes this unique is the coupled interaction between the statistically inferred data collected and the statistically inferred overhead of the process performing that collection, using the latter as a self-tuning feedback mechanism to automatically control the amortized cost of obtaining the former, while also using the data collected by the former to focus and drive the next stage in the process. For instance, the same is done with respect to guiding specialization while governing the amortized overhead of the ongoing specialization subprocess, and again also for bottleneck discovery and relevance weighting, our next topic at hand.

In a broad sense, the idea is to search for and collect weighted expectation data to guide downstream selective tuning efforts toward improving the common cases, while simultaneously statistically monitoring the overhead of this search processes and of the tuning efforts, both. The search naturally advances as data reaches significance limits, enabling the decision points of how to proceed to continue the next phase, from bottleneck discovery to data profiling then on to specialization, re-compilation and dynamic code replacement. At the same time, overhead profiling provides statistical throttling and dampening of this search process as it progresses.

Employing statistical methods both to advance the search process feed forward and to simultaneously throttle the overhead of this search through feedback in a unified way is the third original contribution of this dissertation.

By coupling the two in this natural way, the DESCARTES system is able to balance the effectiveness and efficiency of program optimization in a simple, principled way without resorting to *ad hoc* fixed threshold limits or convoluted control mechanism.

This subtle and elusive interplay of statistical mechanisms I affectionately refer to as *The Strangelove Device* (§ 0.3.0, p. 52).

***PC Sampling* to Discover Bottlenecks.** Up to this point, we've presumed that the bottleneck procedures are the ones being profiled. After all, profiling and optimizing procedures that do not contribute much to the overall running time of a program is wasted effort. This begs the question of how bottleneck procedures are identified in the first place and at what cost are they discovered.

As with *context spectrum* profiling, locating bottleneck procedures could be done through exhaustive profiling of the dynamic call graph. As we've already seen, however, complete profiles are wasteful, both in detail and in running time overhead. Not surprisingly, therefore, I opt once again for statistical sampling as an alternative to exhaustive profiling.

In the case of identifying bottleneck procedures statistically, one records which procedures are invoked over time. As with *context spectrum* profiling, an *opportunity sample* could be employed. Unlike *context spectrum* profiling, however, an additional concern arises.

Namely, large slow procedures that dominate running times should be counted more heavily as bottlenecks than small, short-running procedures that are invoked the same number of times. This argues in favor of weighting the counts of how many times a procedure is invoked by the duration of each procedure invocation. Recording entry times and exit times using a system call to access the process clock to track invocation durations is often done. Invocation counts are then scaled by duration measurements to accumulate an appropriately biased sample of how much time is spent in which procedures. Unfortunately, these system calls to access the process clock can be expensive as they generally require penetrating the language layer to get at the kernel operating system process clock.

Worse, measuring procedure invocation exit times is problematic when a language implements proper tail recursion semantics. For instance, under proper tail recursion, procedures that simply exit directly to their caller's caller's caller could just jump directly to their caller's caller, bypassing the exit/return code of their immediate original caller, hence bypassing the code fragment that would have recorded their immediate caller's exit time. Viewed equivalently (and more traditionally) from the standpoint of the calling procedure at sub-call time rather than from the callee's viewpoint at return time, procedures that end with a direct subroutine call to a procedure, returning directly to their own caller upon its return, can simply jump directly to the target secure in the knowledge that the target procedure will return through the original caller's return address (on the control stack). That is, the caller can replace the usual call-then-return instruction sequence in tail call position with a simple jump instruction (after making appropriate arrangements to pass any arguments expected by the callee, of course). This, for example, led Guy Steele to characterize tail calls as "argument passing `gotos`" [Steele 77]. Exit profiling breaks this.

The above suggests that, perhaps, a procedure should count only the time actually spent in its own local code block, not in calls out to other code. This would require accessing the system process clock before and after each external call in order to accurately account for local-only process time. For highly modular code with lots of calls out to other modules, this could prove prohibitively expensive, both in the time spent punching time clocks and in the space required to instrument all call sites with clock-out and clock-in code.

A far simpler and more elegant solution to this problem of weighting invocation counts with local-only processing time while managing overhead is realized by once again turning to sample statistics. In this situation, however, we cannot resort to successive *opportunity sampling.* Instead, we turn to *periodic sampling*, more commonly referred to by statisticians as *systematic sampling.*

In *systematic sampling*, rather than just sample the first N occurrences of the event of interest as opportunity presents them, one instead samples every i occurrence until the target N number of total instances is acquired. Here we call i the *sample interval* and N the target *sample size.* As before, this proceeds until the sample size is large enough to achieve some target confidence interval on the statistically inferred sample data. By varying the length of the sample interval, we can regulate the overhead of the sample process, just as we varied the delay between successive *opportunity samples* in our earlier discussion.

The benefits of *systematic sampling* over *opportunity sampling* as a means of discovering run-time bottleneck procedures is three-fold.

First, *systematic sampling* is self-biasing so no local duration weight scaling is needed. Specifically, by using a sample interval sufficiently larger than the underlying processor clock, sample data sets will naturally bias themselves with respect to the time spent locally in each procedure since, at each sample interval, we record only which procedure we are currently executing. It doesn't matter where within the local code block we are, all we care about is what was the last code (re)entry point encountered. This eliminates the need to scale sample counts by duration weights since the more time spent locally with any given code block, the higher the probability that that procedure will be sampled at any given sample interval. In sum, periodic sampling is fortuitously intrinsically pre-biased in exactly the way we want.

This, of course, presumes a means of periodically interrupting program flow and reconstructing the last code (re)entry point that led to the current instruction being executed, but that is a fairly straightforward and standard task. It usually goes by the name of *PC sampling* since it samples the value of the "program counter" (or "effective instruction pointer", if you prefer) of the program's underlying run-time virtual machine. In short, periodic *PC sampling* is inherently self-biasing, without fretting over localized duration weight scaling of counts.

As a second benefit, periodic *PC sampling* does not disrupt tail recursion semantics. That is, as a consequence of not having to keep track of invocation duration weights in order to scale sample counts, we don't need to disrupt tail recursive calls to record exit times. In fact, we don't need to disrupt procedure entry points either since we don't compute entry/exit durations at all, at least not directly within the sampled procedures anyway. This altogether avoids requiring language-layer code from calling kernel operating system layer code to access the process time.

In effect, we replace compulsive clock watching with the more relaxed knowledge that, for a sufficiently large number of samples of a given fixed sample interval, we can effectively infer the expected time spent in each procedure as the product of the interval length and the proportion of samples attributed to the chosen procedure. Said more carefully, in the limit, the proportion of *samples* attributed to any given procedure is a good estimator of the proportion of *time* spent in that procedure. So if half the samples are of a procedure, p, we expect about half the total running time over the entire sampling run was spent in procedure p, plus or minus the confidence bands. Note that, in general, absolute measures don't matter for our purposes anyway— only relative proportions are important— so we will rarely if ever have occasion to even compute how much time is spent in any given procedure. We won't care that half a minute is spent in p, just that 80% of our time was spent there in the last sample run, for example.

Our third and final benefit of *systematic sampling* over *opportunity sampling* is that throttling the time spent doing *PC sampling* is a simple matter of dynamically adjusting the *sample interval*. Specifically, each time the interval timer triggers, kernel-level operating system code is launched to service the (asynchronous) timer interrupt. At that time, it is simple and inexpensive for the kernel-level interrupt handler to record entry and exit times of the handler code run to discover and record the current *PC*. This time spent *PC sampling* accumulates to reflect the proportion of time spent in *PC sampler* overhead versus time spent running normal code. When this ratio exceeds some reasonable target percentage, the *sample interval* can by dynamically adjusted accordingly.

In effect, this isolates and centralizes *all* instrumentation and clock punching to a single kernel-level system-layer interrupt handler without requiring *any* changes to (and hence no recompilation of) the subject programs being sampled.

This last point illustrates a recurring theme: by computing the ratio of time spent in profiling, whether it be *PC sampling* or *context spectrum* data profiling, versus the time spent making forward progress in the process at hand, the relative overhead of profiling can be governed to a target percentage of total running time. Put more simply:

> *Self-aware periodic profiling systems can easily self-regulate their overhead by self-tuning their dynamically adjustable sample interval size driven by statistical feedback, without resorting to* ad hoc *system-specific fixed threshold limits.*

This is the key, simple, elegant insight into how dynamically adaptive program specialization can be done efficiently:

> Every profiling mechanism employed should be self-aware and self-tuning to dynamically adapt the overhead of the overall profiling process, using systematic sampling and statistical inference to stay within reasonable target percentage bounds while continually morphing the running code to streamline it for ever-changing patterns of actual usage.

In the DESCARTES prototype, alas, only *PC sampling* is currently done using *systematic sampling*. In a production system, so too would be *context spectrum* profiling as well as continual, intermittent, adaptive *program specialization*, throttling when and how often new specializations are generated and old specializations are re-specialized and/or discarded. Extending

DESCARTES in this way is straightforward so it is left as future work. These shortcomings should not detract from the central proof of concept of the thesis, as the discussion of example experimental results shows in the next few chapters.

Spectral Specialization is the Core Process— revisited. We began this section on original contributions by touting *spectral specialization* as the primary original contribution of this work but we delayed further discussion of that point until we had more closely examined the data structure that drives it, the *context spectrum.* That was the second main original contribution of this work. In passing, we also dovetailed into discussing the third major original contribution, the use of statistical inference as feedback as the means by which the overhead of this entire enterprise can be managed in an efficient and effective way.

Having done all that, finally we are now equipped to revisit how *context spectra* are used to guide *spectral specialization* in the DESCARTES prototype. Specifically, we have yet to investigate how DESCARTES decides what code to specialize, how to specialize it, to what degree and at what cost. We briefly touch on each of these four points in the remainder of this section.

What to Specialize. First, deciding what code to specialize is straightforward: specialize the bottlenecks. As we saw above, *PC sampling* discovers which procedures are bottlenecks.

For example, in the Fibonacci example shown earlier:

```
;;    ----- -----
(define (fib-x/no-ui z)
  (if (< z 2)
      z
      (+ (fib-x/no-ui (-1+ z))
         (fib-x/no-ui (-   z 2)))))
```

A representative sample run produced the *PC sample* histogram of Figure 0-2 (p. 35).

This shows that some 38% of the samples (spec., 2225 of 5808 at a 20 *m*sec sample interval) were attributed to the compiled procedure code block for `fib-x/no-ui`. Some 20% or so were to the `<` system predicate, and so on.

With this many samples, we compute the confidence bands on `fib-x/no-ui` and `<`'s proportions as shown in Figure 0-3 (p. 36).

That is, the proportion of samples attributed to `fib-x/no-ui` was roughly 38.31%±1.25% at the 95% confidence level and 38.31%±1.65% at the 99% confidence level, while `<`'s confidence bands are roughly 19.78%±1.02% at the 95% level and 19.78%±1.35% at the 99% level.

With an initial *specialization threshold* of, say, 20%, only procedures witnessed in 20% or more of the samples at any given moment are deemed sufficiently interesting to warrant potentially specializing, so only those will be spectrally profiled at this time. In this example, it is clearly worth *context spectrum* profiling `fib-x/no-ui`, whereas `<`'s confidence interval straddles the threshold so we will not choose to profile that at this time.

This illustrates that from the *PC sampling* histogram and some initial *specialization threshold*, DESCARTES can automatically decide what code blocks are interesting enough to profile

```
((38.30923 #(2225. 5808. 2225/5808+20i)
           2225.
           (code-block heathen com-proc)
           #[compiled-procedure 41 (fib-x/no-ui "fibs" #x1) #x14 #x14E9C28]
           (fib-x/no-ui z)
           "/sw/ziggy/Projects/Descartes/Fibonacci/fibs.inf"
           1)
 (19.78306 #(1149. 5808. 383/1936+20i)
           1149.
           (code-block purified com-proc)
           #[compiled-procedure 37 (lambda "arith" #xD) #x1AB0 #x2CF43C]
           (&< z1 z2)
           "/usr/local/lib/mit-scheme/SRC/runtime/arith.inf"
           13)
 (12.2073 #(709. 5808. 709/5808+20i)
          709.
          (code-block purified com-proc)
          #[compiled-procedure 39 (complex:-1+ "arith" #x96) #x14 #x2E3828]
          (complex:-1+ z)
          "/usr/local/lib/mit-scheme/SRC/runtime/arith.inf"
          150)
 (10.53719 #(612. 5808. 51/484+20i)
           612.
           (code-block purified com-proc)
           #[compiled-procedure 38 (lambda "arith" #xD) #x16C0 #x2CF04C]
           (&- z1 z2)
           "/usr/local/lib/mit-scheme/SRC/runtime/arith.inf"
           13)
 (10.38223 #(603. 5808. 201/1936+20i)
           603.
           (code-block purified com-proc)
           #[compiled-procedure 40 (lambda "arith" #xD) #x1474 #x2CEE00]
           (&+ z1 z2)
           "/usr/local/lib/mit-scheme/SRC/runtime/arith.inf"
           13)
 (8.78099 #(510. 5808. 85/968+20i) 510. (prob-comp heathen)))
```

Figure 0-2: Fibonacci – PC Sample Histogram

```
Procedure 'fib-x/no-ui':
-----------------------------------------------------------------------
(* 1.96 (sqrt (/ (* 0.3830923 (- 1 0.3830923)) 5808)))  ;;; 95% level
;Value: 1.2502721993694859e-2
-----------------------------------------------------------------------
(* 2.58 (sqrt (/ (* 0.3830923 (- 1 0.3830923)) 5808)))  ;;; 99% level
;Value: 1.6457664665169763e-2
-----------------------------------------------------------------------

Procedure '<':
-----------------------------------------------------------------------
(* 1.96 (sqrt (/ (* 0.1978306 (- 1 0.1978306)) 5808)))  ;;; 95% level
;Value: 1.0245249689307935e-2
-----------------------------------------------------------------------
(* 2.58 (sqrt (/ (* 0.1978306 (- 1 0.1978306)) 5808)))  ;;; 99% level
;Value: 1.3486093978782896e-2
-----------------------------------------------------------------------
```

Figure 0-3: Fibonacci – Confidence Bands

and potentially specialize. By dynamically adjusting this *specialization threshold*, the percentage of overall running time spent specializing procedures can be self-tuned to target a limit percentage.

In this particular *PC sampler* run, only 5% overhead was spent in the *PC sample* itself, so we won't dwell on this issue further at this point other than to note that a 20 *m*sec sample interval seems a fine initial choice, at least on the machine on which this experiment was conducted.

How to Specialize. Earlier, we saw that the *context spectrum* for `fib-x/no-ui`'s single parameter `z` is, in effect, the following (rounding to the nearest decade percentage at the 99% confidence level to simplify the presentation):

`1`	30%
`tiny`	30%
`2`	20%
`0`	20%

To specialize `fib-x/no-ui` on this *context spectrum*, the specializer first considers three things:

1. The current *specialization threshold* (say 20% in this example)...

2. The *statistical relevance* of the presumed bottleneck from *PC sampling* (call it 40% for `fib-x/no-ui` to simplify the presentation)...

3. The input *context spectrum* (as summarized in the above table).

For instance, if the *specialization threshold* were 40% then no procedures would have been spectrally profiled so there would be no candidate procedures to specialize. At this point, the system might adjust the threshold downward if *PC sampling* reveals a high proportion of samples in non-specialized code.

At a 20% *specialization threshold*, `fib-x/no-ui` is spectrally profiled. When its spectral profile accumulates a sufficient number of samples, the worst-case statistical significance of the spectrum will approach a second threshold, the *spectral significance limit* (of say 99%), at which point the profiled procedure becomes a candidate for specialization.

Polyvariant Dispatch. Specialization by on-line polyvariant partial evaluation begins by inspecting the *context spectrum* of the procedure, in this case the heterogeneous spectrum of `fib-x/no-ui`'s single parameter `z`. Only two subparts of that spectrum exceed the 20% *specialization threshold* so only those two variant specializations will be generated at this point. This is done by generating a small, simple variant dispatcher that chooses among the target specializations, falling back to the original non-specialized code if no compatible specialization matches.

For instance, in the `fib-x/no-ui` example, we would generate a specialization of the form:

```
;;        ===========  _____ _____
(define (specialized::fib-x/no-ui z)
  "
  ----------------------------------------
   Spectrum:    z =      1   30%
                      tiny  30%
  ----------------------------------------
  "
  (cond ((exact-positive-one?             z)
         (exact-positive-one::fib-x/no-ui z))     ;;; Expect: 30%
        (( tiny-positive-fixnum?            z)
         ( tiny-positive-fixnum::fib-x/no-ui z))  ;;; Expect: 30%
        (else
         (fib-x/no-ui z))))  ;;; I.e., original non-specialized code
```

Remark: Please forgive the non-traditional use of whitespace. The textual tabular-ization is intended to highlight the inherent symmetry of the code.

This reveals how we break down heterogeneous spectra into dispatch-driven polyvariant specializations for the spectral components. It begs that we next turn to the two simpler specializations, those for `z` equals 1 and for `z` a tiny positive integer.

Sub-Specialization: `exact-positive-one` Variant. For the `exact-positive-one` case, we specialize `fib-x/no-ui` for input `z` of type `exact-positive-one` and expectation 30%. This results in the following:

```
;;        ------------------  _____ _____
(define (exact-positive-one::fib-x/no-ui z) 1)
```

This was done in several steps.

First, the *statistical relevance* (40%, from *PC sampling*) of the call being specialized was scaled by the spectral weight (30%, from data profiling) for this case, resulting in an overall specialization weight of 12% for `z` being `exact-positive-one` within the body of the specialized `fib-x/no-ui` variant dispatch procedure. This information is carried in the *specialization environment* of the on-line partial evaluator (hereafter, "the specializer") as we partially evaluate the body.

```
(if (< z 2)
    z
    (+ (fib-x/no-ui (-1+ z))
       (fib-x/no-ui (-   z 2))))
```

To specialize the `IF` expression, we first specialize the predicate clause, *viz.,* `(< z 2)`. The generic system procedure `<` is an *arity dispatch procedure* that, in this case, branches to the `binary-<` generic two-argument version. That, in turn, is recognized by the specializer as a primitive procedure for which a table-driven specialization is performed. For an `exact-positive-one` first argument `z` (as looked up in the specialization environment) and a second argument of constant 2, this predicate can be reduced directly to `true` since 1 *is* less than 2 and, more importantly, we know that `z` of type `exact-positive-one` can be only 1.

Note that even though the type binding of `z` to `exact-positive-one` was only observational (with relative expectation of 12%), when `z` does have that binding, the predicate is axiomatically `true`, not just probably so. That is, although our global expectation of ever entering this current specialized variant in the first place is only 12%, once we've done so we *know* with certainty that the parameter is an `exact-positive-one` since that is a pre-condition of this specialized variant's call interface. Nevertheless, we carried along the global 12% expectation weight to drive subsequent partial evaluation decisions (like whether to in-line/unfold calls out or just residualize code instead), as we'll see more clearly in the next specialization case (for `z` a `tiny-positive-fixnum`).

To continue, since the predicate clause specialized to constant `true`, the consequent clause of the `IF` expression is specialized in place of the original `IF` form and the alternative branch is ignored.

Proceeding, then, with the consequent clause, we specialize the variable reference `z` in the specialization environment where it is bound to the type `exact-positive-one`. Since this is a singleton type (a unit set), we specialize `z` directly to its sole possible value, the constant 1. This I call *uniquification* or *The Highlander Optimization*,[13] replacing specialized expressions with literal atomic constants where they can only ever attain one unique known atomic value at run time.[14]

[13] After the advertising poster tag line for the movie Highlander [Widen 86], namely: *"THERE CAN BE ONLY ONE"*.

[14] This is a case of replacing instances of *representationally equal* objects with *implementationally identical* objects when the former implies the latter (in SCHEME parlance, when `equal?` implies `eq?`). This is not to be confused with *extensional equality* or *observational equivalence* (in SCHEME parlance, `eqv?`).

Specifically, in axiomatic set theory, *extensional equality* refers to *functions* with identical domain, range and map. It does not traditionally refer to non-function values. Meanwhile, in formal semantics, *observational equivalence* refers to the equivalence of *terms* in all contexts, including those contexts in which side effects

The net effect is that `exact-positive-one::fib-x/no-ui z` is a constant function returning 1. It ignores its parameter `z` but that's OK: we retain the original call interface regardless so this specialized variant can be easily textually substituted for the original in places where that would be appropriate but, for whatever reason, open coding is deemed inappropriate.

Sub-Specialization: `tiny-positive-fixnum` Variant. For the next case, we specialize `fib-x/no-ui` for input `z` of type `tiny-positive-fixnum` and expectation 30%, again under the 40% *statistical relevance.*

This proceeds as follows.

```
(if (< z 2)
    z
    (+ (fib-x/no-ui (-1+ z))
       (fib-x/no-ui (-   z 2))))
```

First, the `IF` expression's predicate clause, `(< x 2)`, is specialized in the extended specialization environment. In this case, `z` is a `tiny-positive-fixnum`, which describes any integer greater than 2 but less than the integer "tiny"-ness limit (whose exact value isn't important here). As before, this means that partial evaluation of this predicate expression ultimately performs a table-driven specialization of the primitive `binary-<` call, in this case to discover that it always specializes to `false` no matter what particular element of the set `tiny-positive-fixnum` is bound to `z`.

Partial evaluation of the `IF` expression therefore proceeds directly with the alternative ("else") branch, ignoring entirely the consequent ("then") clause. This is the symmetric opposite of what happened in the previous case (`z` of type `exact-positive-one`).

Here things get interesting.

Source Code Canonicalization. At this point, the partial evaluator encounters the following code fragment with the current specialization environment and *statistical relevance*:

```
(+ (fib-x/no-ui (-1+ z))
   (fib-x/no-ui (-   z 2)))
```

For clarity, it helps to imagine that this nested expression has been rewritten into a sequence of simple named subexpressions, as in:

---

might occur. Terms which are *observationally equivalent* may be made identical within the store in any given programming language implementation, but this so-called "*intern*"ing is not mandated by the formal semantics.

By contrast, what I dub here *implementational identity* stipulates that the two values denoted are in fact identical: their referents are physically the same object in memory— in short, they have the same address. The larger LISP community thus often refers to this as *pointer equality*.

This precarious quagmire of subtle shades of equivalence/equality/identity ultimately can be traced back to the analytic ontology of von Leibniz [von Leibniz 1686, Section IX], specifically his principle of the *Identity of Indiscernibles*, often cited as simply "*Leibniz's Law*". This according to the Stanford Encyclopedia of Philosophy [http://plato.stanford.edu/entries/identity-indiscernible/].

See also related entries at Wikipedia [http://www.wikipedia.org].

```
;;       ====  ---  _____ _____
(define (flat::tpf::fib-x/no-ui z)
  "
  ---------------------------------------
   Fib[z]  for  z  a tiny positive fixnum.
  ---------------------------------------"
  (let ((fib:z-1 (let ((z-1  (-1+ z)))
                   (fib-x/no-ui z-1)))

        (fib:z-2 (let ((z-2  (- z 2)))
                   (fib-x/no-ui z-2))))
    (+ fib:z-1
       fib:z-2)))
```

This uses `let`-*extraction* to ensure that all operators and arguments are either variables or literal constants. This names all anonymous intermediate results while flattening each procedure call to a syntactic list of atoms. The DESCARTES specializer does this sort of de-nesting internally (and a few other simple transformations) prior to initiating partial evaluation. This kind of source code canonicalization is standard practice, if idiosyncratic per system. Of course, DESCARTES chooses less mnemonic names for newly introduced intermediates (like **Arg_1_0**) so I took the liberty of renaming them here for clarity.

For ease of discourse, however, it is helpful to further transform this into a linear sequence of sub-tasks, as in:

```
;;       ======  ---  _____ _____
(define (linear::tpf::fib-x/no-ui z)     ;;; z is a tiny-positive-fixnum.

  (define     z-1  (-1+ z))  --.
  (define fib:z-1            | <----.
       (fib-x/no-ui z-1))  --'      |  These two pairs
                                    |  can be reordered.
  (define     z-2  (- z 2))  --.    |
  (define fib:z-2            | <----'
       (fib-x/no-ui z-2))  --'

  (+ fib:z-1
     fib:z-2)
  )
```

DESCARTES does not do this transformation: this I do here solely to make the ordering of subtasks explicit for the following discussion.[15]

[15] In fact, strictly speaking, these ordered sequences of DEFINE forms are not proper SCHEME code. This is because standard SCHEME [IEEE 91] does not specify an order of evaluation of internal DEFINE forms. The above works assuming MIT SCHEME's a top-to-bottom/left-to-right order of evaluation of internal DEFINEs. Still, one could easily re-write these instead as sequences of SET!s to locally bound placeholders or as a LET* cascade, but since this is for expository purposes only, that would serve merely to clutter and somewhat obscure the code listing while making it more difficult to isolate each individual form in the flowing text. Clarity trumps fastidiousness in this case.

Continuing using this transformed source, specializing the expression `(-1+ z)` for `z` known to be a `tiny-positive-fixnum` (hereafter "`tpf`") is straightforward, along the lines of how we specialized primitive `<` above. Here the table-driven specialization generates the code:

```
(fix:-1+ z)
```

This calls a special primitive defined only on fixed-point numbers (non-big integers). It is the concrete base primitive that generic `-1+` ultimately bottoms out on for integers within a given safe range. Its details are not important here except to note that `tpf`s are within that range. (By the way, `-1+` is the primitive decrement operator, or unary minus.)

Inference of Specialization Return Types. Now consider how to specialize the subsequent call: `(fib-x/no-ui z-1)`. In particular, what is the type distribution of `z-1` with which to specialize this recursive call to `fib-x/no-ui`?

```
;;       ======  ---  _____ _____
(define (linear::tpf::fib-x/no-ui z)     ;;; z is a tiny-positive-fixnum.

  (define     z-1  (fix:-1+ z))
  (define fib:z-1
         (fib-x/no-ui z-1))      <---- Uh oh:  z-1 has what spectral distribution?

  (define     z-2  (- z 2))
  (define fib:z-2
         (fib-x/no-ui z-2))

  (+ fib:z-1
     fib:z-2)
  )
```

The interesting question here is: what is the type of the result of executing `(fix:-1+ z)` given that `z` is a `tpf`. This is interesting only because `-1+` is not closed on `tpf`s. Specifically, for all but one `tpf`, the result will be a `tpf` but for the smallest `tpf` (*viz.,* 3) the result will be of type `exact-positive-two`.

The interesting question here thus becomes: what is the expected spectral distribution of these two distinct types of results? This is interesting because there are at least three ways to approach answering that question.

Statistical Vacillation. First, since there are two possibilities with no particular way of knowing *a priori* which will occur, we could weigh them equally, expecting a `tpf` result half the time and 2 the other half. At least this way, anything that makes decisions downstream based on these weights will not be unfairly biased either way. This kind of reasoning is called *fiducial inference* in the statistics community. It is conservative in the sense of least commitment, but its use is rare and often controversial. (See [Bulmer 67/79] for a nice discussion.)

A perhaps better expectation of results can be derived by considering the distribution of values within the `tpf` domain. Since there are presumably many such values but only one of them decrements to 2, we would expect a vast majority of the time to get a `tpf` result and only rarely the result 2. This presumes that we have a concrete measure on the size of `tpf`. In fact, we do. This represents inferred probability based on *frequency distributions*, or *probability density functions* if you prefer. This is the standard non-controversial statistical approach.[16]

A final and more definitive approach is to admit that we do not really know in practice what the actual distribution of values of `z` will be other than the constraint of being a `tpf`. For example, it might always be 37 for some random context or it might always be 3 in some other setting. In the former extreme case, the decrement would always yield a `tpf` but in the latter it would always yield 2. Specializing code based on either of these assumptions is therefore plausible, so this might be seen as an argument in favor of *fiducial inference* when we have no idea of the distribution of values of `z`.

To summarize, inferring outcomes based on *probability density functions* presumes a uniform distribution (equal likelihood) on possible input values. As just argued, however, the actual input distribution may be far from uniform in practice. Embracing a principle of least commitment therefore seems prudent since it at least avoids subsequent downstream specialization of code paths that may never arise at the expense of alternate paths that were not expected but could turn out to be dominant.

For this reason more than any other, DESCARTES by default uses *fiducial inference* for return distributions, although the option does exist to use *frequency distributions* instead. Both are implemented in the prototype. Note, however, that these are employed only where and when sufficient call site specific distribution statistics are not otherwise available yet with adequate relative confidence bounds. They are a temporary stop-gap measure only.

Empirical Resolution. Reasoning like this can make seasoned statisticians cringe. Let's step back from the discussion for a moment to reflect on just what problem we're really trying to solve here in the first place.

```
;;      ======  ---  _____ _____
(define (linear::tpf::fib-x/no-ui z)     ;;; z is a tiny-positive-fixnum.

  (define     z-1  (fix:-1+ z))
  (define fib:z-1
         (fib-x/no-ui z-1))     <---- Aha: fib-x/no-ui!

  (define     z-2  (- z 2))
  (define fib:z-2
         (fib-x/no-ui z-2))

  (+ fib:z-1
     fib:z-2)
  )
```

Shifting the focus of the argument slightly, it's true that our encoding of `z`'s distributional type lacks the *frequency distribution* information we seek when inferring the return type of

[16] Bayesian objections notwithstanding.

`(fix:-1+ z)`, but the real question here is what is the distributional type (spectrum) of `z-1`, not `z`. In other words, lack of *a priori* distribution information for `z` is not the real issue here: lack of *a posteriori* distribution information for `z-1` *is*.

But one procedure's output is another procedure's input, so to answer that directly, we could simply profile the continuation at the return point. When in doubt, measure it.

In this particular case, it suffices just to profile `(fib-x/no-ui z-1)`. Since we already have a *context spectrum* for `fib-x/no-ui`, we can use that as a guide to balancing the *inferred spectrum* for `z-1`.

Derived Spectra. Pressing ahead fearlessly, the *inferred spectrum* for `z-1`, using *fiducial inference*, is:

tpf	50%
2	50%

... while the observed *context spectrum* for `fib-x/no-ui` is:

1	30%
tpf	30%
2	20%
0	20%

... then, given that only the `tpf` and 2 cases are possible for `z-1`, only the matching two spectral elements are feasible in the recursive call.

This leaves an adjusted *in situ* filtered spectrum of:

tpf	30%
2	20%

... for the recursive call, `(fib-x/no-ui z-1)`.

This was obtained by filtering from `fib-x/no-ui`'s *observed spectrum* those spectral elements which are not feasible in the *inferred spectrum* of the argument actually being passed at this call site, the result of `(fix:-1+ z)`.

Normally we would re-scale this to match the expected relative proportions of the *inferred spectrum* but since, using *fiducial inference*, they were equally represented, that re-scaling is idempotent in this case.

We do however need to re-scale the filtered result to add up to 100% since this is a local expectation for `z-1`, not a global expectation of an accumulated observed *context spectrum*. That said, we use the following input spectrum for `z-1` passed to the recursive call to `fib-x/no-ui`:

tpf	60%
2	40%

This filtered and re-scaled spectrum, derived by balancing the return value's *inferred spectrum* with respect to its consumer's input parameter's *observed spectrum*, we call a *derived spectrum* since that's shorter to write than "*in situ* filtered, contextually scaled, fiducially inferred, return value spectrum".

Proceeding with parameter `z` bound to this *derived spectrum* in the specialization environment, we can now finally consider specializing the recursive `(fib-x/no-ui z-1)` call in this updated specialization environment with the ongoing *spectral relevance* of 40%. Whew!

Specializing: `fib:z-1`**.**

```
------------------------
Derived Spectrum for z-1

  tpf  60%
   2   40%
```

With a *specialization threshold* of 20% and a *statistical relevance* of 40%, the recursive call to `fib-x/no-ui` on `z-1` would be further specialized for the `tpf` case (with weighted relevance of $60\% \times 40\% = 24\%$) but not for the 2 case (with weighted relevance of only $40\% \times 40\% = 16\%$).

Its variant dispatcher is shown in-lined in the following code fragment:

```
(define fib:z-1  (if ( tiny-positive-fixnum?             z-1)
                     ( tiny-positive-fixnum::fib-x/no-ui z-1)   ;;; 24%
                     (                       fib-x/no-ui z-1))) ;;; 16%
```

This defers to the original non-specialized `fib-x/no-ui` when `z-1` is not a `tpf` because we chose not to spawn a 2 variant at this time.

Note, however, that if a specialization for argument 2 had been available already for some other reason, this could have read:

```
(define fib:z-1  (if ( tiny-positive-fixnum?             z-1)
                     ( tiny-positive-fixnum::fib-x/no-ui z-1)   ;;; 24%
                     (exact-positive-two::fib-x/no-ui    z-1))) ;;; 16%
```

This is still safe but only because the return spectrum from `z-1` on a `tpf` input is a closed partition on `tpf` and 2: the result `z-1` can never be of any other spectral type. No other `ELSE`-like fall back to the original non-specialized `fib-x/no-ui` was needed.

I'll have more to say about this sort of opportunistic closed-partition peephole specialization in a later chapter. I mention it in passing now only to whet the reader's appetite for considering closed versus opened partitions, whether inferred or derived, noting that *observed spectra* are always open.

That is, observed *context spectra* accumulated through data profiling tell us only what has been witnessed so far, not what new heretofore unseen data types might be witnessed in the future: the absence of evidence is not evidence of absence. By contrast, spectra inferred or derived from axiomatic inference over known input types and closed spectra are themselves closed, whether fiducially inferred or otherwise: we know what we know and that's all there is to know.

Specialization Stubs. One final tweek: even though circumstances conspired in this case such that no `exact-positive-two` specialization was generated nor serendipitously found already at this time, that variant specialization might well get spawned in the future. If it does, we may want to then come back and re-specialize `fib:z-1` (within `tpf::fib-x/no-ui`) to pick it up. In general, several other specialized procedures may also exist that had to punt due to the lack of `exact-positive-two::fib-x/no-ui` being spawned when they were created.

As a contingency plan to avoid the eventuality of such re-specialization fire drills, we could go ahead and spawn an `exact-positive-two::fib-x/no-ui` *specialization stub*, like:

```
;;       ------------------  ----- -----
(define (exact-positive-two::fib-x/no-ui z)    ;;; Non-specialized stop-gap stub: temporarily,...
        (                    fib-x/no-ui z))  ;;; ... just call the original non-specialized code.
```

By arranging for stubs to be generated as needed and called from all appropriate specialization call sites that otherwise would have had to punt, this flurry of triggered re-specializations can be avoided. Simply by re-placing this stub's body later when the specializer really spawns it, all amorous suitors who had been waiting patiently to meet it will just magically share the joy.

Better living through indirection.

Specializing: `fib:z-2`. The specialization of `z-2` and `fib:z-2` are similar. Together, they lead to this partial specialization:

```
;;       ======  ---  ----- -----
(define (linear::tpf::fib-x/no-ui z)

  (define     z-1  (fix:-1+ z))  ;;; Expect: tpf or 2 at 50% each

  (define fib:z-1  (if ( tiny-positive-fixnum?             z-1)
                       ( tiny-positive-fixnum::fib-x/no-ui z-1)    ;;; 24%
                       (exact-positive-two::fib-x/no-ui    z-1))) ;;; 16%

  (define     z-2  (fix:- z 2))  ;;; Expect: tpf or 2 or 1 @ 33% each

  (define fib:z-2  (cond ((exact-positive-one?                 z-2)
                          (exact-positive-one::fib-x/no-ui    z-2))    ;;; 15%
                         (( tiny-positive-fixnum?             z-2)
                          ( tiny-positive-fixnum::fib-x/no-ui z-2))    ;;; 15%
                         ((exact-positive-two?                 z-2)
                          (exact-positive-two::fib-x/no-ui    z-2)))) ;;; 10%
  (+ fib:z-1  ;;; Yet to do
     fib:z-2))
```

Notably, the filtered *observed spectrum* for `z-2` passed to the recursive call `fib:z-2` is:

```
  1   30%
tpf   30%
  2   20%
```

...re-scaled to total 100% as:

`1`	37.5%
`tpf`	37.5%
`2`	25%

When weighted by the 40% statistical significance, these spectral elements become 15%, 15% and 10%, respectively, within the specialization of `fib:z-2`. As with `fib:z-1`, we allow the assumption that a specialization for argument 2 is already available.

Notice that, as before, although none of the *in situ* weights on the possible spectral elements exceeds the *specialization threshold* of 20%, this merely means that no *new* specializations for those cases will be generated at this time. That does not preclude the possibility that they be exploited if they already exist, however. It also does not preclude spawning *specialization stub*s.

Finally, this also exploits the closed partition of `z-2` for `z` a `tpf`: it must always be either a `tpf` or 2 or 1, nothing else.

Specialization Fixed Points. Only `(+ fib:z-1 fib:z-2)` still remains to be specialized. Here the issue of inferred return types becomes problematic. In the case of primitive base operators like unary `-1+`, binary `-` with second argument of 2, and the `<` comparison, table-driven result-type spectral inference was straightforward since their arguments were either literal constants or parameters for which a *context spectrum* was bound in the specialization environment. For the subsequent consumption of those results at the inputs of the recursive `fib:z-1` and `fib:z-2` calls, *derived spectra* served to drive the specialization process forward. For the return type spectral inference of the output of these recursive calls to `fib-x/no-ui`, however, some cleverness is required. It is not simply table driven since Fib is a user-defined procedure, not a built-in primitive base operator of known, pre-analyzed input/output type transformation behavior.

Traditionally, type inference for recursive procedures can be done by iterative Kleene fixed point induction. For instance, `fib:z-1` can have whatever type Fib returns on a `tpf` or 2 while `fib:z-2` can have whatever type Fib returns on a `tpf` or 2 or 1. The type of the sum of any two elements of those two sets of possible return types is what Fib itself returns on `tpf` input. The minimal fixed point of this set of recursive equations is the inferred return type spectrum we seek.

This is typically expressed as a set of recursive equations on types. For instance, if we define $T[t]$ to be the set of possible types of the result of applying Fib to an argument n of type t (*i.e.,* $F[n]$'s type is in $T[t]$ when n has type t), then the type returned by Fib on `tpf` input is just the fixed point of the recurrence relation:

```
------------------------------------------------------------------------------
    T[tpf] = { T[tpf] ∪ T[exact-positive-two] } ⊕
             { T[tpf] ∪ T[exact-positive-two] ∪ T[exact-positive-one] }
     for
        T[exact-positive-one] = { exact-positive-one }
        T[exact-positive-two] = { exact-positive-two }
------------------------------------------------------------------------------
```

...where we define "$\bigoplus$" in the obvious way to be the set of possible types that can result from adding together any pair of elements chosen one each from the two input type sets. It's the result spectrum of `+`.

Abstracting out such recursive equations from static program text and solving them is not insurmountable, assuming they always converge, but this can be computationally expensive in the limit. Note that this approach relies solely on the static analysis of program text. For DESCARTES, thankfully, a cheap and convenient alternative exists.

Similar to the argument made before in the cases of `z-1` and `z-2`, determining the return type spectra for `fib:z-1` and `fib:z-2` is not the real goal *per se*. What *is* the goal is ascertaining the input type spectra for the two arguments to `+` so that it can be specialized appropriately. Once again, spectral profiling is our friend.

Even in the absence of any spectral type information for `fib:z-1` and `fib:z-2`, however, it is still possible to specialize the call to `+` to its embedded two-argument variant, `binary-+`. This at least eliminates the arity dispatch of the top-level `+` arity dispatch procedure. Beyond that, however, specialization of `binary-+` would be idempotent due to lack of spectra for its arguments.

Specializer-Initiated Auxiliary Procedure Profiling Requests. To wit, if no *context spectrum* has yet been collected from profiling `binary-+` then we can do no better than replacing `+` with `binary-+` in the body of `tpf::fib-x/no-ui` (that is, without resorting to spectral fixed point inductive type inference). This in effect is arity-only specialization for missing spectra, which frankly the MIT SCHEME native compiler can already do using only simple static syntax analysis.

After doing that small amount of specialization, however, the specializer could request profiling for `binary-+` since it wanted to proceed but had to truncate the specialization task due to lack of data. There are at least three different ways to initiate and process auxiliary profiling requests like this, which we investigate next.

The first attempts traditional *off-line specialization pre-processing* to anticipate auxiliary profiles that might be needed. This misses the mark. The second introduces the idea of *on-line specialization continuations* as a traditional alternative. This comes closer to being viable but it still raises difficulties. The third and final approach introduces the non-traditional (novel) idea of *specialization-driven self-tuning feedback control.* This last approach is supported by the DESCARTES prototype; the previous two are not.

I present this as a progression of straw-man ideas since this allows us to focus on non-obvious pieces of the puzzle incrementally before arriving at the ultimate approach taken in DESCARTES. This motivates that ultimate resolution by illustrating the shortcomings encountered when one tries to simply generalize and extend the more direct, traditional approaches. Jumping headfirst into the ultimate resolution would not be as fluid and illuminating. Such an oblique approach is more effective than an outright frontal assault on the dominant paradigm.[17]

Specialization Pre-Processing. At profile time, one could anticipate the eventual decision to specialize the profiled procedure, scanning its source code for free variables in op-

[17] As amply demonstrated by the signature "*Oblique Order*" tactic of [Friedrich 1780/1999].

erator position and arranging to profile those as well since their spectra will likely be of use when specializing the primary procedure's code body. This is plausible in specific cases but it is unworkable in general.

The problem with this idea is that the operator of a general procedure call (sub)expression may not be statically decidable due to higher-order procedures, fluid bindings and such. Rewriting source code to explicitly name every operator expression (*e.g.,* via `let`-*insertion* as earlier) might seem to help but this is a red herring since merely knowing a procedure's static local alias does not assist the profiler in determining which run-time procedure object to actually profile. The core problem here is mapping static program text into dynamic procedures for profiling since the spectral profiler profiles actual run-time procedures, not static names or source text. In general, therefore, only by invoking the procedure to be profiled can it be determined which of its subexpressions' operators might be useful to profile as well in order to obtain result spectra for intermediate values.

Advocates of *continuation passing style* might recognize this as the desire to profile a procedure's internal subexpression *continuations* in order to collect spectra of intermediate data. For the uninitiated, a *continuation*[18] is a reification of "the rest of the computation" at a given program point, producing a procedure which can be exported from the computation (in order to spawn a dynamic re-entry point) or which can be invoked directly to advance the computation normally. The SCHEME standard defines the procedure `call-with-current-continuation` to expose these implicit underlying continuations as explicitly named first-class procedures (of one argument, the value to pass on to "the rest of the computation" at the place where `call-with-current-continuation` was invoked). Its use, however, is generally discouraged since it is a fairly heavy-weight mechanism, requiring reification of the control stack into the run-time heap to support multiple re-entrant downstream invocations.

As a light-weight alternative, *continuation passing style* (CPS) [Plotkin 75] is a style of (re-)writing code to make these implicit control continuations textually explicit throughout the program without having to wrap calls to `call-with-current-continuation` around every subexpression. This is done by extending the call interface of every procedure to take an explicit continuation argument, using these explicit continuation procedures to advance the computation by explicitly invoking these ubiquitous continuations on the value to be propagated forward.

This all sounds far more complicated that it is, at least from a programming standpoint. Consider, for example, the following straightforward re-write of our now-familiar Fibonacci program into *continuation passing style.* (The algorithm for automating such a re-write is called *CPS conversion.*)

[18] As recounted by John Reynolds [Reynolds 93], an accurate attribution for the original idea of "*continuations*" is problematic. In short, the idea was first presented by [van Wijngården 66] but went largely unnoticed. It was later resurrected by [Mazurkiewicz 71] but not widely publicized until later. Meanwhile, [Morris 70/93] used it in a narrowly circulated manuscript. Finally, Strachey used it in [Strachey 73] then he and Wadsworth popularized it in [Strachey & Wadsworth 74/2000]. To close the loop, Wadsworth cites Mazurkiewicz as inspiration.

Fibonacci *à la* CPS Conversion. For comparison, here is the original (non-*CPS-converted*) definition:

```
;;       ----- -----
(define (fib-x/no-ui z)
  (if (< z 2)
      z
      (+ (fib-x/no-ui (-1+ z))
         (fib-x/no-ui (-   z 2)))))
```

Here is `fib-x` written in *continuation passing style*:

```
;;       ----- ===
(define (fib-x/cps z k_return)
  (</cps           z 2
   (lambda  (z-below-2?)
     (if     z-below-2?
            (k_return z)
           (-1+/cps   z
            (lambda              (z-1)
              (fib-x/cps          z-1
               (lambda       (fib:z-1)
                 (-/cps                  z 2
                  (lambda                (z-2)
                    (fib-x/cps           z-2
                     (lambda        (fib:z-2)
                       (+/cps fib:z-1 fib:z-2
                              k_return))))))))))))

;;; Trivial CPS style primitives:

(define (</cps x y k)
  (k    (<     x y)))

(define (-/cps x y k)
  (k    (-     x y)))

(define (+/cps x y k)
  (k    (+     x y)))

(define (-1+/cps x k)
  (k    (-1+     x)))
```

Notice that, aside from the funky tabular whitespace (intended to make the code a bit more readable), each primitive generic procedure has been transformed into *CPS style* as well. Also, this code had to commit to an explicit evaluation order for the subexpressions `fib:z-1` and `fib:z-2`, which were not so constrained in the original code.

Note that by invoking these continuations in tail call position (*i.e.,* as non-nested subexpressions), the program "returns" by calling the continuation. This converts intermediate return values into formal parameters of the explicit continuation procedures, thereby providing anchor points for profiling. Re-writing programs into *continuation passing style* is a standard technique called *CPS conversion*.

Specialization Pre-Processing - Continued. That said, the approach suggested above for anticipating missing spectra of intermediate values still requires providing some mechanism for exporting these anonymous internal run-time continuations to the profiler, for instance by globally naming them and accumulating a global data structure of profile targets that will be created when the re-written procedure body is re-compiled, *etc.* By globally naming them and exporting them by name, we provide a means of passing them by value to the profiler without having to first run the program, using the compiler instead to expose the target run-time procedures of interest through global naming of the re-factored subexpression continuations. Note that this even handles the case where the operator of a call expression cannot be statically determined since we profile the continuation leading into that dynamic call, not the call itself. Upon continuation entry, the dynamic context of the call parameters will be apparent as too will be the dynamic operator (since, like `let`-*insertion*, *CPS conversion* had the effect that all calls are comprised entirely of named identifiers and literal constants).

In sum, therefore, this approach to profiling potentially all intermediate results requires re-writing the procedure into *continuation passing style*, re-compiling it in this new form, naming the intermediate subexpression continuations and exporting them by name to a target queue to enable spectral profiling on them. In systems that do internal *CPS conversion* already anyway, this is not overly burdensome. In systems that do not do *CPS conversion*, however, this is a lot of work to expend on paths that may in fact never be run.

The DESCARTES prototype does not pursue this approach, as we shall see shortly. This does however lead us one step closer to a more palatable solution.

First, though, we must dig this hole just a little deeper.

Specialization Continuations. Assume now that programs to be profiled are first *CPS converted* and their intermediate continuations globally named as sketched above. Rather than using a pre-pass to prëemptively profile all their internal continuations, however, imagine we wait to see which of these continuations are actually ever exercised sufficiently during calls to the primary procedure being profiled to warrant their being profiled too. For instance, imagine that the specializer nominates internal continuations for profiling as they reveal themselves to be important for supplying otherwise missing spectra of relevant intermediate results.

The question remains what to do when one of these named and exported auxiliary continuations later becomes adequately profiled to some target level of statistical significance to be helpful. One could entirely redo the specialization of the containing principal procedure that gave rise to it being profiled in the first place, but this could be wasteful. This could also result in a substantial amount of code churn since several profiled continuations might be nominated from a single complex procedure body but their spectra may not attain the target level of statistical significance at the same time, so re-specialization might be re-triggered repeatedly as each one's profile converges in succession.

It might be tempting, therefore, to delay specialization of any procedure until all its subsidiary continuations have accrued sufficient spectral detail to allow full specialization. This state may never be achieved, however, if some continuations lie along an execution path that is never exercised. Waiting only for all the already nominated continuations is only slightly better since, as usage patterns shift, some nominated continuations along no-longer dominant branch paths may not be exercised very often anymore, if ever again, so their spectra may never statistically converge. One might call this "*The Godot Gambit*".[19]

An alternative way to proceed the specialization process from a call site where no spectral information is available for the arguments, neither inferred nor observed, would be to residualize the call expression for the time being but also capture the continuation of the specializer itself at that point. Note that this is the continuation of the specializer program, not the continuation of the program being specialized by it. We call these *specialization continuations*. Using these, for example, could allow the specialization to recommence at the point of insufficient information if and when the missing spectral data ever becomes available, meanwhile allowing specialization to complete at least for the partial spectral data already available. This is a common way to implement backtracking search using explicit continuations. In essence, therefore, this approach recasts specialization as an optimizing search problem over a possibly incomplete or dynamically shifting maze of intermediate value spectra.

Alas, there may be several points within the specialization of any given procedure where the spectra of its intermediate values are unknown, so possibly many *specialization continuations* will be spawned. Consequently, each subsequent reactivation of a given *specialization continuation*, once adequately profiled, will spawn additional downstream *specialization continuations*. These *specialization continuations* will naturally nest so that what order they are eventually re-awakened in could significantly impact the quality of the specialized code generated. Note that attempting memo-ization of *specialization continuations* is of limited help here since as new information becomes available, better specialization results will be possible so we really do want to redo the specialization from those points in light of the new (implicit) context spectrum information. How one might throttle and control the overhead of such a tangled web of dynamically weaved continuations leads to new worries that may require yet more sophisticated mechanisms and machinations to overcome.

Additional layers of cleverness could be piled on here in an effort to address these remaining problems, but we defer that as an exercise for the obsessive reader. Instead, suffice it to say that the DESCARTES prototype does not pursue this approach. It simply gets too complicated too quickly to pursue.

Having fostered an appreciation of the issues that arise from these traditional approaches in some concrete detail, we are now equipped to take a step back to explore a more non-traditional alternative. In short, a much simpler resolution is possible by once again appealing to statistical inference, which is the unifying principle of this dissertation.

Specialization-Driven Self-Tuning Feedback. Rather than try to address the remaining concerns above, let's step back from the continuation-based approaches to reconsider

[19] After Samuel Becket's two-act play, Waiting for Godot [Becket 53/55].

the problem afresh in light of our deeper appreciation of the difficulties involved.

First, *CPS conversion* was suggested as part of an approach to get a handle on the *spectral context* of the target program's intermediate values by profiling internal subexpression continuations. From the viewpoint of the specializer, one might characterize this as a *data continuation* since its goal is to fill in missing data, *viz.,* the intermediate value spectra, that the specializer wants.

Second, the idea of the *specialization continuation* was suggested as a means of retaining intermediate control points within the specializer itself to enable restarting the specialization process in flight as these missing data become available. One might characterize this as a *process continuation*, again from the viewpoint of the specializer.

Both of these ideas attempted to capture explicitly those points of incomplete information where the specializer is forced to make a potentially undesirable compromise by residualizing a code fragment that might well benefit from more aggressive optimization if only the statistical information were available to direct it appropriately. "For want of a nail, ...".[20]

"How I Learned to Stop Worrying and Love Statistical Inference." In essence, the core difficulty is that the specializer has available, at specialization time, only the source code text for the program to be specialized as well as a specialization environment mapping each parameter to its respective *context spectrum* and mapping all global free variables to their bindings. This alone does not allow the specializer to know for any given internal operator what its value will be at run time nor what will be the spectra of all its arguments.

Thankfully, there is a pragmatic resolution to this problem. Specifically, even if the operator within some specialized procedure's body cannot be determined statically, the procedure it corresponds to *will* still get specialized *if* it is called sufficiently often to have independently been deemed worthy of profiling and eventual specialization in its own right. So although, in general, the caller will not directly in-line calls out to this specialized dynamic operator, the specialized code for each effective operator will be generated and called by virtue of this independent specialization. If it never rises to the level of relevance to have appreciably improved the specialization, no worries: it won't be pursued. If, on the other hand, it does become pertinent, it will be automatically tuned appropriately on the other side of the call interface (that is, internal to the callee).

Moreover, in the case where an operator *can* be statically identified at specialization time, that operator can be queued for profiling (and thus subsequent specialization) if the *statistical relevance* of the call justifies it. This means that only those auxiliary procedures that clearly would benefit from specialization at some important call site will be added to the queue of procedures to profile and subsequently specialize. In fact, if a priority queue keyed on the call

[20]Americans are most familiar with this proverb as rendered in Benjamin Franklin's Poor Richard's Almanack (1758 Edition) [Franklin 1758], but The Oxford Dictionary of Proverbs traces it back as far as John Bower's *Confessio Amantis* [Bower 1386/1857/1901/1966,1980,2000,2003,2005, Verse 4785]. The particular phraseology specific to the loss of a horse due to want of a nail is attributed (*ibid.*) to a French proverb dating from the late 15th century, therein rendered by Jennifer Speake as: «*par ung seul clou perd on ung bon cheval*», which she translates as "by just one nail one loses a good horse". An expanded version is also included in various renditions of The Tales of Mother Goose.

site's *statistical relevance* is used to nominate potential auxiliary procedure profile targets, the overhead incurred can be throttled and therefore regulated. Finally, the principal procedure that nominates an auxiliary procedure for profiling can be associated in the priority queue with the nominee, and *vice versa*, so that if/when the nominee accumulates sufficient spectral information, the nominating procedure can, in turn, be queued for re-specialization to take advantage of this new information. By queuing procedures (perhaps even by name or by object hash code), the size of the contents of this queue will be small and easily managed compared to the unbounded retention bloat of queuing continuations instead.

If this profile/specialize feedback loop should ever quiesce so that no new procedures are being profiled (and thus specialized) under the current *specialization threshold*, the threshold can be relaxed to include, say, the first few queued nominees in a disciplined and orderly fashion. In short, priority queues keyed on *statistical relevance* provide global rigor and discipline on what otherwise would be a tumultuous tangled web of dynamic interdependencies. A single, simple, global waterline mechanism suffices to govern the whole system. This adaptive waterline trigger mechanism I affectionately refer to as *The Strangelove Device*.[21]

In sum, this design stratagem might be summarized as *Statistical Inference with Feedback-based Tuning*, or "SIFT" (to coin yet another acronym), as a less whimsical alternative to the moniker "*The Strangelove Device*". This term captures the notion that *statistical inference* is used dynamically (at run time) to characterize and quantify the otherwise statically unquantifiable emergent aspects of the *dynamic abstract context* with respect to which a software system can be efficaciously specialized. It also captures the idea that statistics can likewise be employed as well to measure and regulate the overhead of gathering these statistical data histograms (*context spectra*) used to drive and focus this adaptive specialization process.

In this way, SIFT-based program profiling collects and filters the evolving bottleneck usage data (both which procedures are hot and how they are used) to capture the statistically significant dominant workload characteristics of actual program runs, within induced error bounds. Consequently, SIFT-based program specialization can focus, drive and throttle resources (both running time and specialized code space) for perspicacious cost/benefit trade-offs within the identified envelope of statistical expectations of performance improvements based on a sliding window of past experience. Finally, SIFT-based self-tuning of various control parameters (such as the profiling *sample interval*s and *confidence limit*s as well as the *specialization threshold*s) can restrain the overhead of this ongoing streamlining process to be effectively governed within acceptable bounds, at least in the asymptotic long term steady state.

[21] This is an oblique reference to the film Dr. Strangelove [Kubrick 64], subtitled "How I Learned to Stop Worrying and Love the Bomb", a corruption of which heads this subsection. Amusingly, the two reported working titles of the film also seem ominously *apropos*, they being: "A Delicate Balance of Terror" and "Edge of Doom".

The movie parodied the concept of a "Doomsday Device" which, once activated, could not be disabled. It was meant as a deterrent to global thermonuclear war by means of *mutual assured destruction* (a.k.a. MAD) [Kahn 60,69,78,2003,2007]. Oh, the delicious irony of such apocalyptic logic!

I look forward to someday unleashing DESCARTES on itself— to profile the profiler and specialize the specializer and specialize the profiler and profile the specializer and so forth— to see if it self-immolates in spectacular fashion, or just quietly shuts itself down with a whimper. Such experimental self-application is the gold standard within the partial evaluation community [Ruf & Weise 92,93].

This unified method of quantification of dynamic expectations within measured confidence bounds is unique to this project. Other systems may employ pieces of this puzzle to varying extent, but assembling all three components— profiling, specializing and self-tuning— into a uniform whole is a novel contribution to the field, which distinguishes this approach from related work.

Alternatively, a more encompassing overall term might be "Statistical Inference-driven Specializations Yielded by Perspicacious Histograms with Ubiquitous Self-tuning", or simply SISYPHUS,[22] in recognition that the task of dynamically adaptive specialization is perpetual: its work is never done.

Yet I prefer "DESCARTES",[23] as reflected in the title of this dissertation and the name of the prototype as proof of concept of its thesis.

Still, those averse to acronyms may prefer to dub this simply *Cartesian program specialization* rather than *Sisyphean specialization* or SIFT-*based specialization* or what have you.[24] Regardless of the preferred moniker, it is hoped that this work stands on its merits as both innovative and efficacious. "A rose by any other name...".[25]

Taken altogether, this allows the statistical inference of relative relevance and accumulated significance to naturally drive re-specialization based on the on-demand profiling of auxiliary procedures that appear, in practice, to be worthy of additional optimization, as witnessed by the specializer. This avoids *ad hoc* limits and thresholds typically employed in these sorts of feedback trigger mechanisms. It also avoids tempests of re-specialization and their resultant avalanches of dynamic continuation generation and re-activation. In short, it just works.

Moreover, in the long-term steady-state. the overall profiling/specialization system can quiesce to a very low-overhead maintenance state.

Specifically, if periodic PC sampling detects no new non-specialized procedures with run-time relevance above the specialization threshold and if no existing specializations are detected to be running significantly above their former specialization-time relevance, then no new data profiling will be initiated by the procedure profiler. At the same time, as each data-profiled procedure eventually accrues a sufficient number of samples to cross the statistical confidence threshold for its accumulated context spectrum, data profiling is suspended. Finally, when no new data profile spectra are forthcoming, no new pending specialization tasks will be *en queue* (and *vice versa*).

When the overall system has reached this level of inactivity for a sufficient number of successive sample windows, only PC sampling will be still active, at which point the sample period can be gradually lengthened to reduce DESCARTES' overhead to some minimal watchdog main-

[22] After Albert Camus' essay, «Le mythe de Sisyphe: essai sur l'absurde», translated as The Myth of Sisyphus, essay on the absurd [Camus 42,85/55/75,2000,2005].

[23] In obvious *hommage* to René Descartes' supremely self-aware proclamation: *Cogito ergo sum*: "I think therefore I am" [Descartes 1637/1978,1983]. The rumor that his last words were "I think not" has never been substantiated. Somehow, I doubt it.

[24] *Suum cuique.* « *Chacun son goût.* » *Etc.*

[25] [Shakespeare 1594, Romeo & Juliet, Act II, Scene 2, Lines 43–44].

tenance state.[26] At that point, the overall overhead of DESCARTES can be tuned down to some arbitrarily low user-specified target level, trading steady-state overhead for responsiveness in detecting— and therefore adapting to— subsequent characteristic changes in workload behavior, including brief bursts of anomalous behavior signifying procedural phase changes or data working set transitions. In fact, using DESCARTES as a statistical mechanism to detect and formally characterize such state transitions in actual workloads should prove a compelling area of future work. Existing tools seem unequal to the task.

[26]Noting, however, that the overhead of PC sampling alone was below 5% with even the smallest possible sampling interval on a modern mid-performance development machine, the need for tuning down the PC sampling interval to enter a "*deep sleep*" mode seems a bit obsessive.

This, for example, was obtained on a modest 1.67 GHz AMD ATHLON XP-2000+-based desktop machine with 1 GB of DDR core memory running REDHAT LINUX 7.3 atop the LINUX 2.4.20-37 kernel, using MIT SCHEME 7.6.1 LIAR (Intel i386) 4.114 with a 9000-block heap (max). This is by no means a high-performance machine by modern standards. Nonetheless, future kernels will eventually reduce the minimal sample interval for timer interrupts to keep pace with increasing processor clock speed, so the need for "*deep sleep*" retreat will likely come and go as clock speeds and kernel timer resolutions lead and lag one another.

On the original machine on which early prototype data was collected, for example, the PC sampling overhead could be as high 30% when set to the minimal interval. That was considered a powerhouse workstation in its day, so the same issue is likely to come and go in cyclic fashion, following *Moore's Law* as usual. That original machine, by the way, was a 50 MHz PA-RISC PA-7100 HP 9000 series 715/50/64 workstation with 64 MB of 60ηsec RAM core memory running HP-UX 10.10, using MIT SCHEME 7.4.2 LIAR (HP-PA) 4.106 with a 5000-block heap (max). Though a hot commodity in its day, it's system characteristics are laughable by today's standards. Such is the dictate of *Moore's Law*.

Interestingly, the comparative results on the two platforms were virtually indistinguishable: although the profiling overhead improved markedly, the relative speed-up factors of the resulting specializations were nearly identical. « *Plus ça change, plus c'est la même chose.* »

Finishing Fibonacci. In light of the preceding discussion, unless and until `binary-+` is itself profiled sufficiently to produce a useful *context spectrum* of its own, we can imagine the specialization of `fib-x/no-ui` to result as in Figure 0-4 (p. 56).

```
;;       ======  ---  ----- -----
(define (linear::tpf::fib-x/no-ui z)

  (define     z-1  (fix:-1+ z))  ;;; Expect: tpf or 2 at 50% each

  (define fib:z-1  (if ( tiny-positive-fixnum?            z-1)
                       ( tiny-positive-fixnum::fib-x/no-ui z-1)   ;;; 24%
                       (exact-positive-two::fib-x/no-ui    z-1))) ;;; 16%

  (define     z-2  (fix:- z 2))  ;;; Expect: tpf or 2 or 1 @ 33% each

  (define fib:z-2  (cond ((exact-positive-one?               z-2)
                          (exact-positive-one::fib-x/no-ui    z-2))   ;;; 15%
                         (( tiny-positive-fixnum?            z-2)
                          ( tiny-positive-fixnum::fib-x/no-ui z-2))   ;;; 15%
                         ((exact-positive-two?               z-2)
                          (exact-positive-two::fib-x/no-ui    z-2)))) ;;; 10%

  (binary-+ fib:z-1  ;;; Further specialization deferred
            fib:z-2))
```

Figure 0-4: Specialization of `fib-x/no-ui` up to `binary-+`

In a run of Fib[39] where the *specialization threshold* is lowered to 10%, `binary-+` yields the spectrum of Figure 0-5 (p. 57).

```
(#(#(#[compiled-procedure 35 (binary-+ "arith" #xD) #x1474 #x2CE300] -->
     #[lambda 34]                                                    ==>
     (binary-+ z1 z2))
   #(z1
     =
     #(hetero-type
       #(102334154
         ((exact-positive-one    61.8034  . 63245986)
          ( tiny-positive-fixnum 23.5961  . 24146871)
          (exact-positive-two    14.5898  . 14930352)
          (small-positive-fixnum   .01069 .    10943)
          (      positive-fixnum  0.      .        1)
          (      positive-bignum  0.      .        1)))
       ()
       ()))
   #(z2
     =
     #(hetero-type
       #(102334154
         ((exact-zero            38.1966  . 39088169)
          (exact-positive-one    38.1966  . 39088169)
          ( tiny-positive-fixnum 14.58319 . 14923587)
          (exact-positive-two     9.01699 .  9227465)
          (small-positive-fixnum   .00661 .     6763)
          (      positive-fixnum  0.      .        1)))
       ()
       ())))
 .
 102334154)
```

Figure 0-5: Spectrum for a run of Fib[39] with 10% *specialization threshold*

It is important to note that each base type is disjoint from all others. Specifically, “tiny” integers exclude “small” integers which exclude “large” integers which exclude “bignums”. They are range-bounded partitions. This enables us to mix and match their dispatch predicates without fear of overlap.

Resuming specialization with this new information, we summarize the formerly missing spectrum for `binary-+` as follows, rounding to the nearest half decade percentage:

z1		z2	
1	60%	0	40%
tpf	25%	1	40%
2	15%	tpf	15%
:		2	10%
:		⋮	

Note that, as usual, we disregard spectral elements below the *specialization threshold* (in this case, 10%). It is also useful, for presentation purposes, to round the expectations to the nearest half threshold (5%).

Unfortunately, these per-parameter spectra are not cross correlated. Fibonacci aficionados will recognize that when `z2` is 0, `z1` is always 1, since the `z1` and `z2` being added here are Fib[$n-1$] and Fib[$n-2$], but this spectrum does not encode that. Had the spectra been collected as a single spectrum over the pair of parameters rather than as two independent spectra over the individual parameters, this correlation would be reflected in the spectrum.

This refinement would be tantamount to collecting the spectrum of the formal parameter list as if it were a manifest list, not an enumeration of independent elements. Extending this to also include the free variables (in some canonical order, such as depth first order of first appearance within the text of the procedure body) in essence elevates the implicit nominal *dynamic data set* into a manifest ephemera for purposes of spectral snapshotting. Although the DESCARTES prototype does not presently implement this refinement, it might be a useful natural extension for future work.

That the DESCARTES prototype does not collect correlated spectra as described above is an unfortunate oversight. This forces it to consider, in effect, the entire weighted cross product of the individual parameter spectra.

```
----            ----           --------
z1              z2             z1 × z2
----            ----           --------
 1    60%
                 0    40%       24%      ≈ 25%
                 1    40%       24%      ≈ 25%
                tpf   15%        9%      ≈ 10%
                 2    10%        6%      ≈  5%
                 ⋮
tpf   25%
                 0    40%       10%      = 10%
                 1    40%       10%      = 10%
                tpf   15%        3.75%   ≈  5%
                 2    10%        2.5%    ≈  1%
                 ⋮
 2    15%
                 0    40%        6%      ≈  5%
                 1    40%        6%      ≈  5%
                tpf   15%        2.25%   ≈  1%
                 2    10%        1.5%    ≈  1%
                 ⋮
 ⋮
```

To specialize `binary-+` for this spectrum and *statistical relevance* of 40%, we consider those pairings of `z1` × `z2` for which their conjunctive expectation (product weight), scaled by the situational relevance of 40%, is greater than the *specialization threshold* of 10%. That means only those with unscaled product of 25% or higher will be considered (since 40% × 25%=10%).

This leaves only the following candidate spectrum:

```
----            ----           --------
z1              z2             z1 × z2
----            ----           --------
 1    60%
                 0    40%       24% ≈ 25%
                 1    40%       24% ≈ 25%
                 ⋮
 ⋮
```

Recalling that the `z1` and `z2` being added here are, respectively, Fib$[n-1]$ and Fib$[n-2]$, we see that this specializes for the two base cases of the Fibonacci series, which account for nearly half of the total calls to `fib` in the tree recursive exponential algorithm. These statistical expectations reflect exactly that, as observed by profiling.

Alas, for `z` a `tpf` (*i.e.,* 3 or higher), Fib$[n-2]$ will never be 0, so this is a wasted case for the `+` inside the `tpf` specialization variant.

This candidate spectrum leads to the following disciplined final specialization:

```
;;      ====== ---  _____ _____
(define (linear::tpf::fib-x/no-ui z)

  (define     z-1  (fix:-1+ z))  ;;; Expect: tpf or 2 at 50% each

  (define fib:z-1  (if ( tiny-positive-fixnum?             z-1)
                       ( tiny-positive-fixnum::fib-x/no-ui z-1)   ;;; 24%
                       (exact-positive-two::fib-x/no-ui    z-1))) ;;; 16%

  (define     z-2  (fix:- z 2))  ;;; Expect: tpf or 2 or 1 at 33% each

  (define fib:z-2  (cond ((exact-positive-one?                z-2)
                          (exact-positive-one::fib-x/no-ui    z-2))    ;;; 15%
                         (( tiny-positive-fixnum?             z-2)
                          ( tiny-positive-fixnum::fib-x/no-ui z-2))    ;;; 15%
                         ((exact-positive-two?                z-2)
                          (exact-positive-two::fib-x/no-ui    z-2)))) ;;; 10%

  (if (exact-positive-one? fib:z-1)            ;;; Expectation: 60% × 40% = 24%
      (cond ((        exact-zero? fib:z-2) 1)  ;;;  1 + 0 = 1 @ 25% × 40% = 10%
            ((exact-positive-one? fib:z-2) 2)  ;;;  1 + 1 = 2 @ 25% × 40% = 10%
            (else
             (1+                  fib:z-2)))   ;;; i.e., generic 'inc' fall back
      (binary-+           fib:z-1 fib:z-2)))   ;;; i.e., generic 2-arg fall back
```

Note that since the parameter spectra are in this case observational (not axiomatically deduced), they reflect open partitions on the sets of possible types. For that reason, the ellipses in the summarized candidate spectra become generic calls to `binary-+` in the `ELSE` clauses of the specialized code. In the case where `fib:z-1` is known to be 1, moreover, this can be further specialized as `(binary-+ 1 fib:z-2)` by *uniquification* (discussed above, p. 38), which in turn can be further specialized into `(1+ fib:z-2)`, exploiting SCHEME's built-in increment primitive procedure. The important point to note, however, is that this `1+` is fully generic, as is `binary-+`.

This is critical for safety, preserving the semantics of the original code, since these arguments *will* overflow into non-tiny `fixnum`s for even modest initial arguments of `z`. In fact, 39 was chosen as the initial argument to profile for this example precisely because that is the smallest integer argument whose result overflows into the `bignum` representation on the machine implementation in which DESCARTES was developed. It is a good boundary condition test case.

Facts notwithstanding, and for completeness, here is how the blissfully ignorant might have specialized `binary-+` as a straight cross product of the full spectra (so effectively with a 0% *specialization threshold*):

```
(cond ((exact-positive-one?    fib:z-1)                ;;; Expectation: 60%
       (cond  ((         exact-zero?      fib:z-2) 1)  ;;;  1 + 0 = 1 @ 25%
              ((exact-positive-one?      fib:z-2) 2)  ;;;  1 + 1 = 2 @ 25%
              (( tiny-positive-fixnum?  fib:z-2)
               (fix:1+                   fib:z-2))    ;;;              10%
              ((exact-positive-two?      fib:z-2) 3)  ;;;  1 + 2 = 3 @  5%
              (else
               (1+                       fib:z-2))))
      ((tiny-positive-fixnum?  fib:z-1)                ;;; Expectation: 25%
       (cond  ((         exact-zero?      fib:z-2) fib:z-1);;;         10%
              ((exact-positive-one?      fib:z-2)
               (fix:1+         fib:z-1))               ;;;              10%
              (( tiny-positive-fixnum?  fib:z-2)
               (fix:binary-+   fib:z-1  fib:z-2))     ;;;               5%
              ((exact-positive-two?      fib:z-2)
               (fix:2+         fib:z-1))               ;;;               1%
              (else
               (     binary-+   fib:z-1  fib:z-2))))
      ((exact-positive-two?    fib:z-1)                ;;; Expectation: 15%
       (cond  ((         exact-zero?      fib:z-2) 2)  ;;;  2 + 0 = 2 @  5%
              ((exact-positive-one?      fib:z-2) 3)  ;;;  2 + 1 = 3 @  5%
              (( tiny-positive-fixnum?  fib:z-2)
               (fix:2+                   fib:z-2))    ;;;               1%
              ((exact-positive-two?      fib:z-2) 4)  ;;;  2 + 2 = 4 @  1%
              (else
               (2+                       fib:z-2))))
      (else
       (binary-+               fib:z-1  fib:z-2)))    ;;; i.e., generic 2-arg fall back
```

...where `fix:2+` and `2+` are defined in the obvious ways:

```
(define (fix:2+ z) (fix:1+ (fix:1+ z)))  ;;; Double increment presumed...
(define (    2+ z) (    1+ (    1+ z)))  ;;;  ... to be faster than (+ x 2)
```

or

```
(define (fix:2+ z) (fix:+ z 2))          ;;; Double increment presumed not...
(define (    2+ z) (    + z 2))          ;;;  ... to be faster than (+ x 2)
```

This has the disadvantage, among other things, of following a failed test of 1 for `fib:z-1` in the first, outermost `COND` predicate with a test for `fib:z-1` a `tpf` in the second, outermost predicate, which has only a 25% expectation of being true. Had a *weighted* cross product been used instead, the second test would instead be on `fib:z-2`, not `fib:z-1`, since `fib:z-2` has a 40% expectation of being 0 independent of `fib:z-1`'s type.

For comparison, therefore, the *weighted* cross product approach yields code of the following form:

```
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
60%:        1   60%
                           0   40%     24%   ≈   25%
                           1   40%     24%   ≈   25%
                         tpf   15%      9%   ≈   10%
                           2   10%      6%   ≈    5%
                           ⋮
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
40%:                       0   40%
          tpf   25%                    10%   ≈   10%
            2   15%                     6%   ≈    5%
            ⋮
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
40%:                       1   40%
          tpf   25%                    10%   ≈   10%
            2   15%                     6%   ≈    5%
            ⋮
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
25%:      tpf   25%
                         tpf   15%      3.75% ≈    5%
                           2   10%      2.5%  ≈    1%
                           ⋮
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
15%:        2   15%
                         tpf   15%      2.25% ≈    1%
                           2   10%      1.5%  ≈    1%
                           ⋮
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
15%:                     tpf   15%    ≤ 0.15% ≈    0.1%
            ⋮≤ 1%
          ----          ----          --------
          z1            z2            z1 × z2
          ----          ----          --------
10%:                       2   10%    ≤ 0.10% ≈    0.1%
            ⋮≤ 1%
```

This is arguably better since, for any dynamic path through this code, the next predicate tested after each failing predicate is the next most likely to succeed. This improves the average expected time to execute this two-way dispatch.

In general, an n-argument specialization dispatcher generates better code by treating the n uncorrelated argument spectra as independent priority queues, weighted by expectation, from which to always test the most likely of the n arguments' types first rather than generating a blind cross product. Again, the best strategy is to collect correlated parameter spectra to begin with.

One final point in passing that may have occurred to the LISP-savvy: some languages (notably, COMMON LISP and its various dialects) provide a special `typecase` form that dispatches on known machine types. It can do so by extracting the manifest type tag from a datum then dispatching on the choice of literal types explicitly enumerated in the `typecase` dispatch form. Since each type tag is a known language literal, this can be done by simply hashing the extracted type tag and doing a constant-time dispatch without having to otherwise traverse the cases in incremental order. It is an error for the same type tag literal to appear more than once in the case clauses, for example.

SCHEME provides a similar mechanism in the general `case` form, where each literal case clause can be matched against the result of a general expression, including an expression to extract the underlying type tag from the target match term.[27]

This stratagem works well when all type matches are to disjoint manifest type tags. In the general case of *spectral type*s, however, the set of disjoint base types is *not* simply the set of the underlying native implementation's manifest type tags. They instead require range checks on the actual values. For instance, 1, 2, "tiny", "small" and "large" integers all share a common `fixnum` type tag, but they are range-bounded (as noted earlier, p. 57). Thus mere manifest type tag dispatch is not sufficient for full *spectral type* dispatch in most cases: the range checks must still be performed, in order.

This issue becomes even more pertinent were we to generalize our notion of "type" to include general user-defined predicates (*predicate dispatching* [Ernst, Kaplan & Chambers 98]), such as the generalized "type" predicates `Mersenne-prime?` and `Affine-polynomial?`, each of which have important applications in the areas of cryptography and linear systems, respectively. This consideration becomes still more prevalent should we blur the definition of "type" to encompass enumerations of concrete values (for example, to support *value profiling* [Calder, Feller & Eustace 99]).

We'll revisit this topic later, in the chapter on *context spectra* (Chapter **??**, p. **??**). For now, it suffices to note that specialization driven by *context spectra* can exploit the embedded expectation statistics within the *context type*s to produce polyvariant dispatch stubs that are optimized for the observed common cases.

In closing this section, the following is the full result of specializing Fib using the above ideas

[27] MIT SCHEME, for example, sports a (machine-specific) `object-type` primitive predicate that extracts the underlying type tag and returns it as an integer (`fixnum`) encoding of the object's manifest type, which can be used as a match literal in `case` clauses. The MIT SCHEME compiler uses this to great effect throughout the run-time system, for example.

as well as a few simple tricks discussed in more detail in the chapter on *spectral specialization* (Chapter **??**, p. **??**). These tricks include *uniquifying* singletons (mentioned earlier), promoting constant tests above range tests, eliding subsumed adjacent cases, absorbing subsumed `ELSE` code redundancies, demoting rare cases to subsequent `ELSE` fall backs, reducing terminal predicates of closed partitions to `ELSE` clauses, propagating predicate consequence constraints down conditional branches, employing monotonicity and domain closure inference, and such... all of which should be relatively obvious in context for the following code.

Without further ado, here are the original Fib and its fully specialized result given the above candidate spectrum using this weighted cross product dispatch.

First the dispatch trampoline:

```
;;       ==== ---- ________
(define (flat-hack-spec-fib           z)       ;;; Entry: (100.0000) relevance
  (cond ((strict-exact-positive-one?   z) 1)            ;;; (30.9017)
        ((strict-exact-positive-two?   z) 1);;CONSTANT  ;;; (19.0983)
        ((strict-exact-zero?           z) 0);;CONSTANT  ;;; (19.0983)
        ((strict-tiny-positive-fixnum? z)               ;;; (30.9017)
         (tpf::fib-hack                z))
        (else        (flat-generic:fib z))));; I.e., unspecialized original fib-x
```

And now the `tiny-positive-fixnum` variant specialization:

```
;;                    ==== ________          ;;-------------
(define            ( tpf::fib-hack tpf:z) ;; Invariant: z has type strict-tiny-+-fixnum
                   (%tpf::fib-hack tpf:z));;-------------
(define-integrable (%tpf::fib-hack tpf:z)       ;;; Entry:  (30.9017) relevance
  ;;-------------
  ;; Invariant:  tpf:z has type strict-tiny-+-fixnum  (precondition for entry)
  ;;-------------
  ((named-lambda (tpf::fib-hack:+          z1  z2);;; z1 = F[z-1] & z2 = F[z-2]
     (cond ((strict-exact-positive-one?    z1    )      ;;; 61.8034 - (19.0983)
            (cond ((strict-exact-positive-one? z2)      ;;; 38.1966 - (11.8034)
                   2) ; 61.8034 X 38.1966 - (X 30.9017) --> 23.6068 - ( 7.2949)
                  ((strict-exact-positive-two? z2);[CNST];;  9.0170 - ( 2.7864)
                   3)                                   ;;;  5.5728 - ( 1.7221)
                  ((small-positive-fixnum?     z2)      ;;; 14.5838 - ( 4.5066)
                   (fix:1+                     z2))     ;;;  9.0133 - ( 2.7853)
                  (else
                   (int:1+                     z2))))
           ((strict-exact-positive-one?        z2)      ;;; 38.1966 - (11.8034)
            (cond ((exact-positive-two?    z1    );[CNST];; 14.5898 - ( 4.5085)
                   3) ; (38.1966 X 14.5898) - (X 30.9017) =  5.5728 - ( 1.7221)
                  ((small-positive-fixnum? z1    )      ;;; 23.6067 - ( 7.2949)
                   (fix:1+                 z1    ))     ;;;  9.0170 - ( 2.7864)
                  (else
                   (int:1+                 z1    ))))
           ((strict-exact-positive-two?    z1    );[CNST];; 14.5898 - ( 4.5085)
            (cond;((strict-exact-positive-two? z2);[CNST];;  9.0170 - ( 2.7864)
               ;;; 4)                                 ;[MERE];  1.3156 - (  .4065)
               ;;;((small-positive-fixnum?     z2) ;[MERE]; 14.5838 - ( 4.5066)
               ;;; (fix:2+                     z2));[MERE];  2.1277 - (  .6575)
                  (else
                   (int:2+                     z2))))
           ((strict-exact-positive-two?        z2);[CNST];;  9.0170 - ( 2.7864)
            (cond;((small-positive-fixnum? z1    ) ;[MERE]; 23.6067 - ( 7.2949)
               ;;; (fix:2+                 z1    ));[MERE];  2.1286 - (  .6578)
                  (else
                   (int:2+                 z1    ))))
           ((small-positive-fixnum?        z1    )      ;;; 23.6067 - ( 7.2949)
            (cond ((small-positive-fixnum?     z2)      ;;; 14.5838 - ( 4.5066)
                   (fix:+                  z1  z2))     ;;;  3.4428 - ( 1.0639)
                  (else
                   (int:+                  z1  z2))))
        ;;;((small-positive-fixnum?            z2)      ;;; 14.5898 - ( 4.5085)
        ;;; (int:+                         z1  z2))     ;;; Subsumed by ELSE clause
           (else                                        ;;;    .0107 - (  .0033)
            (int:+                         z1  z2)))) ; 4.5085+0.0033=( 4.5118)
  ⋮
```

```
⋮
;; Note: Use specialization cache to in-line 'flat-hack-spec-fib' here!
((named-lambda (tpf::fib-hack:F_z-1          z-1);; Split 30.9017 in half:
   (if (strict-exact-positive-two?           z-1)          ;;; (15.45085)
       1                             ; Expect: 50. X 30.9017 = (15.45085)
       ;;-------------
       ;; Invariant: z-1 has type strict-tiny-+-fixnum (only other possibility)
       ;;-------------
       ;;    (int:+  (<call_global> (<global> flat-hack-spec-fib)
       ;;                           (fix:-1+ z-1))
       ;;            (<call_global> (<global> flat-hack-spec-fib)
       ;;                           (fix:-2+ z-1)))
       (tpf::fib-hack z-1)))         ; Expect: 50. X 30.9017 = (15.45085)
 (fix:-1+ tpf:z))                    ; Two or tiny-positive-fixnum (50/50)
;; Note: Use specialization cache to in-line 'flat-hack-spec-fib' here!
((named-lambda (tpf::fib-hack:F_z-2          z-2);; Split 30.9017 in thirds:
   (cond ((strict-exact-positive-one?        z-2) 1);[CONSTANT];; (10.30057)
         ((strict-exact-positive-two?        z-2) 1);[CONSTANT];; (10.30057)
         (else;(strict-tiny-positive-fixnum? z-2)            ;;; (10.30057)
          ;; (int:+  (<call_global> (<global> flat-hack-spec-fib)
          ;;                        (fix:-1+ z-2))
          ;;         (<call_global> (<global> flat-hack-spec-fib)
          ;;                        (fix:-2+ z-2)))
          (tpf::fib-hack z-2))))
 (fix:-2+ tpf:z))))                  ; Two or one or tiny-+-fixnum (1/3 each)
```

The only two subtle tweaks this includes that have not already been discussed involve:

0) replacing `binary-+` by `int:+`, `int:1+` and `int:2+`
1) omitting the zero test on `fib:z-2`

The former requires deducing that, for `z` a `tpf`, both recursive calls to `tpf::fib-hack` are closed on the integer domain. This involves a simple and safe Kleene fixed point type induction on base types: there are a small finite number of arithmetic base types so this induction always and quickly converges. Given that `binary-+` can be replaced by `int:+`, further specializing it for cases where an argument is 1 or 2 is obvious.

The latter is a bit more complex so it is left as a mystery until the next chapter.

It is also interesting to note that there are a few branches through this code that will never be exercised, though the non-correlated `+` spectrum does not reveal that. For instance, when either `fib:z-1` or `fib:z-2` are 1, the other cannot exceed 2, so it is never a `tiny-positive-fixnum`. Second, `fib:z-2` can never be larger than `fib:z-1` so, specifically, when `fib:z-1` is 1, `fib:z-2` cannot be 2. Finally, `fib:z-1` and `fib:z-2` can never both be 2 at the same time. These are properties of the Fibonacci series that are beyond the reach of simple profiling.

Tantalizingly, if we raise the *specialization threshold* to 10% unscaled (that's 3% scaled by relevance), all these infeasible paths evaporate, being absorbed into their respective `ELSE` fall-back clauses. Unfortunately, this also eliminates the specific specialized case for `fib:z-1` of 2 and `fib:z-2` of 1, changing it from the constant 3 to the slightly less optimal `ELSE` clause fall-back clause: `(int:1+ fib:z-1)`. So raising the *specialization threshold* can eliminate unrealizable specialization paths but it may also leave some feasible ones less specialized as well.

Moreover, by not specializing branches with a "mere" weighted conjunctive expectation below 1%, for example, the entire outermost clauses for `fib:z-1` or `fib:z-2` of 2 are absorbed into their respective `ELSE` clauses, calling `int:2+` as appropriate. Since the joint expectation of these paths being exercised is so small, we wouldn't expect the effort involved in testing for them to be justified in practice, let alone the space they consume in extra snippets of code that should rarely execute.

The above specialized result shows these "mere" cases as commented out code. It does not use a 10% unscaled threshold to eliminate infeasible paths, though.

This is nearly twice as fast as the original and around twice the size when compiled to native machine code (thanks in large part to rampant code sharing).

Last, please note that this is exactly how *not* to write a maintainable, efficient implementation of Fibonacci. This is the kind of code we want our run-time system to automatically spawn on the fly and use internally (possibly saving somewhere in persistent storage for later perusal and for posterity). It is not the kind of brittle kludge that humans should have to produce. Ever.

Compare it with the simple, obvious original code.

```
;;       ----- -----
(define (fib-x/no-ui z)
  (if (< z 2)
      z
      (+ (fib-x/no-ui (-1+ z))
         (fib-x/no-ui (-   z 2)))))
```

The remaining chapters of this dissertation detail just how this specialization was generated, as well as a couple of other interesting examples.

The next chapter, for instance, is a detailed walk through the three case study experiment programs: Fibonacci, Pedigree and Diffusion.

The chapters after that isolate the individual components of DESCARTES for more formal and thorough consideration.

0.3.1 Overview of Experimental Results

Experiments using this prototype have demonstrated that a handful of carefully chosen principled specializations can dramatically improve code speed with only moderate profiling overhead and modest code-space growth. For example, profile-driven automatic optimization of a program to analyze genetic pedigrees (as Bayesian belief nets) has produced a speedup factor of 5 over the original highly optimized code produced by the MIT C SCHEME compiler. A thermal diffusion simulation was sped up by a factor of 8, and the tree- recursive Fibonacci program was sped up by a factor of 2. These were achieved with less than a 2-fold increase in code size and below 5% overhead in execution time for profiling.

0.3.2 Overview Summary

This dissertation explores the design space for effective profile-driven adaptive program specialization, under the practical constraint that the overall system must be efficient with respect to the dynamic overhead.

Specifically, it details how the DESCARTES prototype decides what code to optimize, how to optimize it, to what degree and at what cost. It conclude by considering how this prototype system could be used as the foundation for a fully automated, continual, on-the-fly, dynamically adaptive program optimization system.

0.4 Dissertation Outline

"Is there nothing better for you to do then?", Nullary asked.

"There are plenty better things to do," replied The Tinker.
"...just nothing I'm better at doing," he added, forlornly.

— MARCUS ROSS, *The Many Varied Adventures of Nullary Nonesuch*

0. Context: Introduction, Motivation, Overview, Outline (*i.e.,* this chapter)
1. Experimental Results Walk Through: Fibonacci, Pedigree and Diffusion
2. Procedure Profiling: Bottleneck Discovery through *PC Sampling*
3. Data Set Profiling: Breakpoint Parsing using *Context Spectra*
4. Spectral Specialization, Re-Compilation and On-the-Fly Code Replacement
5. Closing the Loop: Statistical Inference as Feedback Control
6. Summary: Conclusions, Related Work, Future Work, *etc.*

   Bibliography

   Appendices

   Index

   Glossary (maybe)

End of Chapter 0.

[This page intentionally left very nearly blank.]

[This page intentionally left very nearly blank.]

Part O

Prologue

Part I

Examples

[This page intentionally left very nearly blank.]

Part II

Details

Part III

Conclusion

[This page intentionally left very nearly blank.]

Part IV

Appendices