Proc. Int'l. Workshop on Memory Mgmt., St Malo, France, Sept. 1992, Springer LNCS 637.

NREVERSAL of Fortune[1] — The Thermodynamics of Garbage Collection

Henry G. Baker

Nimble Computer Corp., 16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

Abstract

The need to *reverse* a computation arises in many contexts—debugging, editor undoing, optimistic concurrency undoing, speculative computation undoing, trace scheduling, exception handling undoing, database recovery, optimistic discrete event simulations, subjunctive computing, etc. The need to *analyze* a reversed computation arises in the context of static analysis—liveness analysis, strictness analysis, type inference, etc. Traditional means for restoring a computation to a previous state involve checkpoints; checkpoints require time to copy, as well as space to store, the copied material. Traditional reverse abstract interpretation produces relatively poor information due to its inability to guess the previous values of assigned-to variables.

We propose an abstract computer model and a programming language—Ψ-Lisp—whose primitive operations are injective and hence reversible, thus allowing arbitrary undoing without the overheads of checkpointing. Such a computer can be built from reversible conservative logic circuits, with the serendipitous advantage of dissipating far less heat than traditional Boolean AND/OR/NOT circuits. Unlike functional languages, which have one "state" for all times, Ψ-Lisp has at all times one "state", with unique predecessor and successor states.

Compiling into a reversible pseudocode can have benefits even when targeting a traditional computer. Certain optimizations, e.g., update-in-place, and compile-time garbage collection may be more easily performed, because the information may be elicited without the difficult and time-consuming iterative abstract interpretation required for most non-reversible models.

In a reversible machine, garbage collection for recycling storage can always be performed by a reversed (sub)computation. While this "collection is reversed mutation" insight does not reduce space requirements when used for the computation as a whole, it does save space when used to recycle at finer scales. This insight also provides an explanation for the fundamental importance of the push-down stack both for recognizing palindromes and for managing storage.

Reversible computers are related to *Prolog*, *linear logic* and *chemical abstract machines*.

Introduction

Those behind cried "Forward!"
And those before cried "Back!"
T.B. Macaulay, *Lays of Ancient Rome — Horatius* (1842)

A physics revolution is brewing in computer science because many of the abstract models traditionally used have failed to provide deep insight into parallel and distributed computation. Discrete-time serial automata cannot faithfully model a relativistic world in which communication is more expensive than computation. Standard Boolean AND/OR/NOT logic found in all modern computers generates too much heat for use in high-performance 3D logic circuits. Parallel imperative programs have proved to be a nightmare to debug. Lattice-based compile-time analysis has reached its limits, yet significant problems in "aliasing/sharing", "strictness/laziness" and "resource estimation" remain.

Physicists, on the other hand, routinely decide deep questions about physical systems—e.g., they can talk intelligently about events that happened 15 billion years ago. Computer scientists retort that computer programs are more complex than physical systems. If this is true, then computer scientists should be embarrassed, considering the fact that computers and computer software are "cultural" objects—they are purely a product of man's imagination, and may be changed as quickly as a man can change his mind. Could God be a better hacker than man?[2]

Computation has heretofor been based on *writing* metaphors, either the *chalkboard* metaphor—e.g., the von Neumann model—or the *pen-and-paper* metaphor—e.g., the functional/logical model, rather than the *mechanical* metaphor of physics. The use of writing metaphors is curious, since a large fraction of computation is devoted to the simulation of physical systems!

The property that makes the writing metaphor so attractive is that *reading* has very little cost compared with *writing*; something can be read many times without alteration, and this property can be used to *broadcast* information for

[1]Apologies to Alan Dershowitz.

[2]E.g., none of the Ten Commandments is concerned with von Neumann's Machine, nor does Moses mention ever seeing von Neumann on the mountain where the Commandments were obtained.

inexpensive reuse at a number of different times or locations. This property allows physical simulations to be instrumented in discreet ways not allowed by Heisenberg's principle.

Mechanical systems do not have the advantage of inexpensive copying, because mechanical objects can be in only one place at one time—they are conserved. While information can be recorded in an arrangement of mechanical objects and this arrangement can be copied into a similar arrangement of similar objects, the expense of this copying can be calculated and may be quite large. The mechanical metaphor would thus seem to be at a hopeless disadvantage compared with the writing metaphor for general purpose computing.

We have enjoyed the advantages of cheap copying so long, however, that we have difficulty perceiving its penalties. Its most apparent disadvantage is that software companies have a difficult time getting paid for their software—a significant irritation, but not important enough to dramatically change the future of computing. Illegal cloning, however, demonstrates a major problem with cheap copying—which of the copies is the "real" one, and how can multiple copies be kept consistent? "Object-oriented computing", a current major software trend, has at its core an assumption of *object identity*, which is the ability of a software object to have an identity which distinguishes it from other objects. In physical terms, the identity of an object must be *conserved*, meaning that it has a particular location which is different from the locations of other objects. According to Alan Kay [Kay77], it is no accident that object-oriented programming began with the *Simula* language for (physical) simulation.

A major problem in the programming of software simulations is in assuring that the simulations are faithful to the "real" system—i.e., whether the software model obeys the same conservation laws as the "real" system. If the real system is a physical system, then the conservation laws are the laws of physics—e.g., the conservation of mass, energy, etc., while if the real system is an economic one, then the conservation laws are the laws of economics—e.g., the conservation of money. Much of the complexity of modern software systems can be traced to these requirements—e.g., file backup and recovery, transaction models, type systems (including those for physical units), etc. For example, if one physically removes a physical file from a physical filing cabinet, it no longer resides in the filing cabinet, so that there is no possibility of conflict. The reading of a *software* file, on the other hand, does *not* remove the file, so we must go to extra trouble to avoid conflicts with writers and other readers. In short, we must use an lot of computing "machinery" to simulate an innately conservative "mechanical" system.

Current computer *languages* are based on ideas from 3 models—the von Neumann random-access memory, Church's lambda calculus, and Boolean logic networks. The serial nature of von Neumann RAM's shows up in conditionals, goto's, and assignments; this model has been attacked by advocates of the functional/applicative/logic languages based loosely on a write-once policy, which allows for more parallel execution while preserving Church-Rosser determinism. Boolean circuits can be claimed by both the functional/logical and the RAM communities, depending upon whether they are purely combinational or have feedback. All of these models, however, assume 1) that the *duplication* of information is free; and 2) that the *destruction* of information is free.[3] A RAM reads a cell many times between writes, knowing that the value will always be that of the last write; the S combinator cheerfully copies; and Boolean circuit models may allow indefinite fanout. A RAM write wipes out the previous memory contents; the K combinator knowingly kills; and Boolean AND's and OR's are not invertible.

Physical circuits have never lived up to these ideals. Physical storage (cores, DRAM's) must be refreshed, fan-out is limited and too much heat is generated. Yet the physical systems of which real computers are composed do not have these problems. All microscopic physical processes are inherently *conservative*, which means that in addition to conserving mass, energy, momentum, etc., they also *conserve information*. The fundamental theorem of mechanics states that "phase space is incompressible" [Penrose89], which means that two separate "states" cannot converge, nor can a single state diverge. Of course, separate points which are initially "close" in phase space may become widely dispersed; this behavior has been termed *chaotic*, even though chaotic mechanical systems still conserve information. It is this theorem that produces the limits on a Carnot heat engine, because for a Carnot engine to produce *work*, it must somehow *summarize* the information which describes the randomness in a high temperature reservoir using a smaller amount of energy in the lower temperature reservoir. It is not the excess heat *energy* that must be exhausted into the lower temperature reservoir, but the excess *information*! Since the reliability of encoding this information is inversely proportional to the temperature, one can encode this information using less energy only when the exhaust temperature is lower.[4]

Electrical engineers spend a great deal of their time modelling traditional non-conservative two-state Boolean logic with reversible physical processes, which are most uncooperative. Thus, one must switch hundreds or thousands of electrons, and give up hundreds or thousands of "kT's" to make sure that a single bit is transmitted reliably. If we

[3]The "garbage collector" can erase a "write-once" memory in functional systems without contradiction, because the memory has previously become inaccessible.

[4]Black holes seem to "eat" information; perhaps black holes are the "cooling fans" for the Biosphere of the universe.

want to utilize these devices to compute conservative computations, we must then spend hundreds or thousands of devices to reliably simulate conservatism. In other words, we have given up factors of 10^6-10^9 in time and/or energy dissipation for nothing! Yet functional/logic programming, database programming and many other computations (e.g., FFT's) either are, or can readily be reorganized to be, conservative! Biological information processing, for example, dissipates far less heat/information. The processes used to copy DNA and transcribe RNA are mostly conservative, or else the heat dissipated during cell mitosis in an embryonic chicken would produce a hard-boiled egg!

If heat dissipation were the only problem with traditional computer models, the hardware circuits underlying individual functional units such as multipliers and caches could be re-implemented using conservative logic and thereby eliminate 80-90% of the wasted heat. We would then not need to change the abstraction seen by the programmer or compiler; after all, computer engineers have been hiding the physical truth from programmers for over 50 years.

Yet it is actually at the *highest* levels of computer usage where the traditional computer models are wearing the thinnest. One finds it very difficult to ensure that an accounting system "conserves money", or to ensure that a sort program "conserves records", or that a file system "preserves information". It is difficult to program a compiler to optimize the use of registers in the face of arbitrary control structures, or for parallel process to automatically "back down" during an optimistic concurrency control. Static analysis has failed to routinely produce important information, such as that needed for "escape analysis", "strictness analysis" and "aliasing/sharing analysis". For example, Milner's elegant type inference algorithm is routinely used to statically decorate ML programs with type information, which is useful for partial correctness, but type information produces at most an order of magnitude performance improvement over a dynamically typed system. Milner's algorithm can also be used to produce deeper structure sharing information [Baker90UC], but we conjecture that this usage will bring out the algorithm's latent exponential behavior. In short, static analysis techniques can produce interesting information only when their computational complexity is hopelessly exponential.

As a result of these considerations, we advocate the use of models of computation which have more structure because they obey more laws and restrictions. Strongly typed and functional systems are restrictive in their own way; we advocate restrictions which inherently conserve information—i.e., copying is expensive, and information destruction is impossible. Since information is conserved, these programs are *reversible*; reversibility becomes a property as important as the determinacy guaranteed by the Church-Rosser theorem. Because deciding reversibility for bulk computations is generally unsolvable, we guarantee it instead by constructing computations from reversible primitives which are reversibly composed.

While many computations are obviously reversible—e.g., FFT's—a skeptic might wonder how often this is true. For example, a sorting algorithm apparently throws away at least n•logn bits of information in the process of sorting n records. However, if the input records are given "serial numbers", then the original order can be retrieved by resorting on the serial numbers. Since the concatenation of serial numbers to the input records is conservative, we find that it is the *erasure* of these serial numbers at the end of sorting that inhibits reversibility, and not the sorting operation itself. The Newton iterative square root algorithm $x_{i+1}=(x_i+N/x_i)/2$ exemplifies a large class of computations in which the output appears to be independent of the input, since the Newton iteration produces the square root independent of the initial approximation. Nevertheless, if this algorithm is run a fixed number of iterations on infinite precision rational numbers, and if $x_0 \geq \mathrm{sqrt}(N)$, it is *reversible*![5] Newton's iteration essentially encodes the initial conditions into lower and lower order bits, until these bits become insignificant. It is the *erasure* of these low order bits through rounding and/or truncation that dissipates the information needed for reversibility. In a sense, Newton's iteration converges only because Newton's inverted iteration produces chaos.

Many arithmetic operations such as increment and negate are invertible. Multiplication, however, opens up the possibility of multiplying a number by zero and losing all information about the number. In this instance, we must have a "multiplication by zero" exception which is analogous to "division by zero". Although multiplication is mathematically commutative, we need check only one argument for zero, because we will then know the other argument was zero in the case of a zero product.

Time warp [Jefferson85] was the first general scheme for reversing an arbitrary distributed computation. However, the primitive message-passing actors upon which time warp is based are traditional state machines; therefore, they

---

[5]Curiously, inverting Newton's square root iteration itself requires taking a square root. However, the inputs for these roots will always be *rational perfect squares* if the initial forward approximation is rational. If the initial approximation is chosen so that x_0^2-N is *not* a perfect square, then reversal can use this property as its stopping criterion, and the algorithm becomes reversible even without the requirement for a fixed number of iterations.

can only be "rolled back" if they have saved copies of previous states. If, on the other hand, these primitive actors are all themselves reversible, then most of the clutter can be removed from this elegant scheme.

It is no accident that there is a significant overlap between those working on garbage collection and those working on "undoing" computations for purposes such as nondeterministic search, debugging, backup/recovery and concurrency control. However, the approach previously taken has been to utilize garbage collection to help manage the data structures used for "undoing". Our approach is exactly the opposite—we propose using "undoing" to perform garbage collection. In other words, the concept of reversibility is too important to be left to garbage collection and storage management. Garbage collection by undoing is more powerful than traditional garbage collection, because it can be used even for Actor systems in which the garbage may be quite active and thus hard to catch [Baker77].

Garbage Collection

Garbage collection is a favorite topic of researchers; the number of published papers on this topic approaches 1,000. Yet, the necessity for "garbage collection" in a symbolic processing system has troubled researchers from the beginning. That the symbolic computations required in Artificial Intelligence (AI) applications generate "garbage" which must be recycled, has long been a fundamental assumption; AI applications tend to "hypothesize and test", so the structures created during hypothesizing must somehow be recycled when the test is not successful.

Despite its apparent popularity, garbage collection has never been a subfield of computer science in its own right, because it is always seen as a semantics-preserving *transparent optimization* in a programming language implementation. Since it is by definition "invisible" to the programmer, the topic itself continues to be "swept under the rug". Unfortunately, the rug has developed a huge mound beneath it, because we cannot characterize on theoretical grounds the costs and benefits of various garbage collection algorithms, but must rely on relatively crude and *ad hoc* measurements. For example, we have no theoretical model which can tell us that "generational" garbage collection is inherently better than non-generational garbage collection, even though a number of measurements seem to bear this out.

Whether a computational object is "garbage" or not depends upon one's point of view. From the point of view of the computation itself—i.e., the "mutator"—a garbage object becomes inaccessible and cannot influence the future course of the computation. From the point of view of the system—"mutator" plus "collector", however, the object is still accessible, and will be reclaimed by the collector.

Since it is the mutator which "loses" objects, and the collector which "finds" them, we have another characterization of the mutator/collector interaction. What do we mean by "losing an object"? Clearly, the object is not completely lost, because the collector can find it; however, the mutator has "lost track" of it. More precisely, the mutator destroys its *information* about locating the object, and it is up to the collector to *regenerate* this information. Intuitively, the mutator is a "randomizing" influence, while the collector is an "ordering" influence. Interestingly enough, the mutator and the collector must both be either dissipative, or both conservative; one cannot be dissipative while the other remains conservative. If the collector is conservative and the mutator non-conservative, then the collector must somewhere dissipate the information it computes about which objects are garbage, since it creates a free-list with zero entropy/information. Similarly, if the mutator is conservative, then it doesn't produce garbage, and the collector is trivially conservative.

The efficiency of a generational garbage collector comes from its ability to recover storage before the mutator dissipates the information. It can do this because it can localize easily recovered garbage to a small fraction of the address space. If the newest generation consists of 1/256'th of the cells, and if half of these cells are garbage, then the *temperature* of this generation is the number of cells divided by the amount of information $\approx (N/256)/(N/256) \approx 1°$. The generational collector will have an input "temperature" of $\approx 1°$ and an exhaust "temperature" of $\approx 0.5°$. A non-generational collector collecting the same amount of garbage would have an input "temperature" of $\approx (N)/(.0204N) \approx 49°$, and an exhaust "temperature" of 511/512'th of that. Thus, a generational collector would be operating at a temperature ratio of 2:1, while a non-generational collector would be operating at a temperature ratio of 1.002:1. Since the efficiency of a "Carnot" GC is proportional to the ratio of input to exhaust temperatures, it is easy to see why the generational GC is more efficient. (Of course, no reasonable system would operate a non-generational collector with such a small temperature differential.) Thus, the effectiveness of a generational collector depends upon reclaiming a higher fraction of cells than a full GC—i.e., a "dying" cell is statistically more likely to be young than old. If this is not true, then generational GC will not be effective.

Reference Counting and Functional Programming

The lambda calculus and combinators can be implemented using reference counting to collect "garbage", because these models do not require directed cycles. Many implementations do use directed cycles, however, for "efficiency" in implementing the Y combinator used to express recursion. However, even with these Y combinator loops, reference counting can still be used to collect garbage, because the Y loops are well-structured [Peyton-Jones87].

Functional languages cannot "overwrite" a storage location, because assignment is prohibited. This property has been called the "write-once" property of functional languages, and this property is shared with "logic languages"—e.g., Prolog. Since storage is never overwritten, the entire history of the computation is captured, and it would seem that we are very close to Charles Bennett's original thermodynamically efficient Turing Machine [Bennett73], which kept track on a separate tape of all of its actions. However, standard functional language implementations do not preserve the information regarding which branches were taken.

Interestingly, functional language implementations depend upon garbage collection to implement all of the optimizations which require "side-effects". For example, MLNJ [Appel90] does frame allocation on the garbage-collected heap, so that all garbage normally recycled by "stack allocation" is performed by the garbage collector. Furthermore, the use of shallow binding for the implementation of functional arrays [Baker91SB] allows the garbage collector to perform the in-place "assignment" normally expected in an imperative language implementation.

Since functional programming is a "good thing", and since reference counting is sufficient for these languages, it would seem that little more can be said. However, evidence is starting to mount that the lambda calculus and combinators, which are pure models of *substitution* and *copying*, may not be the best models for the highest performance parallel computers. Although one reason for this is that *copying* may be a very expensive operation for a physical computer, the major reason for the current interest in models which perform less copying—e.g., "linear logic"—is the fact that a system which copies in an unrestricted fashion also loses track of things, and requires a garbage collector.

Conservative Automata and Reversible Computation

A number of researchers have been studying the ultimate limits to computation from a physical perspective in order to guide engineers in designing high performance computers. In particular, these researchers have attacked a major problem for these computers—the generation of heat. As computers become faster and smaller, the removal of heat becomes a very difficult problem. One of the limitations on the number of active devices on a chip, for example, is the ability of the chip package to remove the heat fast enough to keep the chip from immolating itself.

It was long conjectured that heat was an essential byproduct of computation, but Bennett obliterated this conjecture by demonstrating a thermodynamically reversible computer [Bennett73]. Because it is thermodynamically reversible, any excess energy given off during one portion of the computation would be reabsorbed during another portion of the computation, leaving the entire computation energy and entropy neutral. Bennett's results show that computation in and of itself does not generate waste heat, but any *erasure* of information must necessarily generate waste heat.

Fredkin and his collaborators [Toffoli80] [Margolus88] are refining Bennett's work to show models for reversible logic which could be used on a thermodynamically-efficient logic chip. They have exhibited techniques for logic design using *conservative logic*, which conserves entropy by remaining reversible. They go on to show physics-like conservation principles, as well as logical analogues to *energy*, *entropy* and *temperature*.

We were intrigued by their simulation of traditional AND/OR/NOT Boolean logic circuits using reversible logic. In this simulation, each of the traditional gates is mapped into a configuration of reversible logic gates, but these gates have a number of "garbage" outputs, in addition to the simulated outputs. Bennett's key idea in making the emulation reversible is to mirror the mapped circuit with a reversed version of itself, with each "garbage output" being connected to the mirror image wire thus becoming a "garbage input" for the reversed circuit. So far, we have "much ado about nothing", since the output of the reversed computation is exactly the same as the original input! However, we can put additional reversible gates *in between* the mapped circuit and its reverse, which can "sense" the output to either affect some other computation, or to copy the final answer. Thus, just as the reversible computer collects its garbage bits with a reversed computation—after suitably summarizing the computation's result, we will utilize a reversed computation to recycle other kinds of resources—e.g., storage and processors.

Reversible Pointer Automata

A pointer automaton can be constructed from Fredkin's conservative logic using his universal simulation for non-conservative logic [Barton78] [Ressler81]. This simulation is not the most efficient use of logic or space, however. We describe a reversible pointer automaton which is a better model for a conservative higher level language.

Our reversible pointer automata will retain the finite state control and pointer registers, but with some restrictions. We need to make sure that each instructions can be "undone", and must therefore retain enough information about the previous state. All of the instructions are *exchanges*, conditional exchanges, and primitive arithmetic/logical operations. Arithmetic/logical instructions only come in forms that are invertible; e.g., replace the pair of registers which encode a double-precision integer dividend by an integer quotient and an integer remainder, with no change in the case of a zero divisor. Since the double-precision dividend can be reconstituted from the divisor, quotient and remainder, the instruction is invertible.

The following are primitive machine operations, where x,y,z,… denote distinct registers, x:y denotes a double-precision "register" constructed from the concatenation of x and y, and a,b,c,… denote distinct constants. Some operations have additional constraints which cause them to "stick" in the same state. Arithmetic operations on "integers" are performed modulo 2^{w}, where w is the word size.

x <-> y; exchange x and y.
x <-> CAR(y); exchange x and CAR(y).
x <-> CDR(y); exchange x and CDR(y).
y := CONS(x,y) & x := nil; push x onto y and set x to nil.
if x=nil then swap(y,z); conditional exchange operation.
inc(x,y), semantics: x:=x+y; inverse is dec(x,y).
dec(x,y), semantics: x:=x-y; inverse is inc(x,y).
minus(x), semantics: x:=-x; inverse is minus(x).
mpy(x,y,z), 0≤x<z, semantics: x:y:=y*z+x; inverse is div(x,y,z).
div(x,y,z), 0≤x<z, semantics: x:=(x:y) mod z ‖ y:=((x:y)-((x:y) mod z))/z; inv. is mpy(x,y,z).
rot(x,c) rotates the bit pattern in x by c bits; inverse is rot(x,-c)=rot(x,w-c).
xor(x,y), x,y distinct registers, has the semantics x:=x xor y; inverse is xor(x,y).
reverse(x) reverses the bits in register x; inverse is reverse(x).
macro for push(r1,r2): {car(free)<->r1; r2<->cdr(free); r2<->free;}
macro for pop(r2,r1):
  if consp(r2) and null(r1) then {r2<->free; r2<->cdr(free); car(free)<->r1;}
macro for push(r): {push(r,stack);}
macro for pop(r): {pop(stack,r);}

This machine language is carefully designed in such a way that all operations are invertible. This property requires restrictions on the topology of the program flowchart. We can "test" a value in a register, but the value must be preserved during, or restored after, the operations depending upon the test, so that when the program is reversed, we can test it again to reverse the appropriate arm of the conditional. Suitably structured programs can be statically checked for proper topology. Suzuki has proved [Suzuki82] that under these conditions 1) all reference counts are preserved, and 2) garbage cannot be created. Since all cons cells on the free list start out with a unity reference count, they always have a unity reference count—i.e., we have a Linear Lisp. Overall reversibility thus depends upon reversible primitives reversibly composed.

Programming Style

An operation P will have a "right" inverse when there exists an inverse operation P' which undoes the first operation—i.e., P•P'=I (the identity). For example, `COPY` and `EQUAL` are essentially inverses of one another, as are "destructuring bind" and "backquote". We may not be able to execute P' in an arbitrary state, however, because it may have conditions on it which cannot be satisfied. When P' is attempted in a situation in which its preconditions fail, we say that P' *sticks*, without doing anything else. Unlike most models, we do not map sticking states into an overall computational *failure*, because failure loses all information about what failed and how it failed, which is the whole point of reversible computation used for debugging.

The strangeness of the reversible programming style is due mainly to our lack of experience with it. Every subcomputation must be reversible, which means that no information is destroyed within the subcomputation. Loops in which the number of iterations is *a priori* fixed are trivial to reverse; other loops may require a counter to remember the number of iterations.

The hardest part of inverting an iterative program is to arrange that the loop iterations themselves preserve information. While a loop can always store its information in the form of a "trail" (analogous to the Prolog "trail"), the amount of storage required may grow quite rapidly. The ability to summarize the information within a fixed number of "state variables" is the reversible computer's analogue to the "tail recursion optimization", which transforms recursive functional programs into iterative imperative programs.

A reversible computer can utilize the following optimization for loops. Most loop tests rarely succeed, and hence produce little "information"; they cannot be dispensed with, however, else the loop would never terminate. On a reversible computer the loop body can be iteratively doubled in size, and when it has exceeded the final value, the loop can be backed up to the correct point. This scheme involves only O(logn) tests instead of O(n), but the non-test work may double. As a result, it is useful only if the test is expensive relative to the body.

Input is handled by saving the input stream as in many other reversible and functional systems [Barton78] [Jefferson85]. Output could conceivably be "taken back" when reversed, as in pure *Time Warp* [Jefferson85], or saved until there is no further possibility of reversing, and then output, as in standard *Time Warp*.

Ψ-Lisp — Reversible Linear Lisp[6]

In a companion paper [Baker92LL], we introduced *Linear Lisp*, which is a variant of Lisp in which every cons cell has a unity reference count. In this section, we introduce a variant of Linear Lisp called Ψ-Lisp which looks very much like traditional Lisp, but all of its programs are reversible. The primitive operations of Ψ-Lisp are many of the operations discussed in previous sections. We make extensive use of swapping operations, as they are self inverses.

Lambda-expressions in Ψ-Lisp appear identical to those in traditional Lisp; they have a "lambda-list" of formal parameters, and a "body" consisting of a sequence of expressions. There are differences, however. The lambda-expression is considered to have a single "argument" which is a list "destructured" by the given lambda-list, in the manner of ML. All of the variable names appearing in the lambda-list must be distinct, and any reference to a free variable inside a Ψ-Lisp lambda-expression is the *only* reference to the variable to avoid violations of linearity.

The interpretation of a Ψ-Lisp lambda-expression is that the "variables" occurring in its lambda-list are "new" local variables distinct from any other variables which are bound to the respective portions of the argument list, and the cons cells from the argument list itself are returned to the free list during destructuring. The binding of these new variables is reversible, since each is a new variable which had no previous value, and the original argument list can be trivially recovered from the values of the distinct variables. The body of the lambda-expression consists of a list of Ψ-Lisp expressions, each of which is individually reversible, and which can be reversed by reversing the execution of each of the computations in the reversed list. Any changes to the values of the lambda-list variables during the execution of the body will be reversed during reversed execution, thus reconstituting their values.

<u>It is an error if any lambda-list variable still has a value at the end of the execution of the lambda-expression body</u>, because linearity promises that every variable will be accessed *exactly once* within the body, and any such access returns the variable to its unbound state. In other words, <u>the argument list to a lambda-expression is completely consumed during its evaluation</u>; the list itself is consumed during binding, and each of the bound values is then consumed during the execution of the body. This property, along with the other properties of Ψ-Lisp, allows us to conclude that <u>all of the information necessary for the reverse execution of the lambda expression is encoded into its returned value(s)</u>. Similarly, it is an error if any expression in the body—except for the last—returns a value; i.e., these expressions must all be operations with only side-effects, but not values. Of course, these side-effects are all completely local, since linearity forces each free variable to belong to only one closure; i.e., any "local" variables of an enclosing lambda which are referenced by a subsidiary lambda closure can no longer be accessed by the enclosing lambda.

A `let`-expression evaluates an expression and destructures it to bind several new distinct variables (destructuring means never having to say `CAR` or `CDR` again). The body of a `let` expression is a sequence of expressions, in which only the last can return a value. Like a lambda-expression, it is an error for a `let`-expression to terminate while its bound variables still have values. These rules allow the reverse execution of a `let`-expression in the same way they allow the reverse execution of a lambda-expression.

Nested expressions have an interesting interpretation. In general, a particular variable may "occur" only once. The values of sub-expressions are conceptually bound to new intermediate variables, so we could have decomposed nested expressions into an equivalent nest of `let`-expressions. The values of all of the intermediate variables are immediately consumed by the expression in which they are nested, so that they satisfy the restrictions on `let`-expressions. A nested expression is thus executed in a manner analogous to a dataflow architecture in which the values ("tokens") flow through the variables ("wires") and into the application nodes ("operators"); a variable without a value is analogous to a wire without a token. Multiple arguments may be evaluated in parallel [Baker92LL].

The most difficult and interesting case is that of the if-then-else expression. This expression has 4(!) sub-expressions—a Boolean test expression, a Boolean predicate, and two arms, a then-arm and an else-arm. If we were to execute the test expression in the normal fashion, then any arguments passed to it would be consumed, and it would be difficult to remember the direction of the evaluation during execution reversal. Therefore, we will evaluate the test expression somewhat differently from the expressions of the arms. We evaluate the test expression to determine the direction of the execution, and then *we undo the test expression to restore the values of any of its variables* before executing the appropriate arm. The variables referenced in the test expression must be referenced again in both arms, since otherwise they would not be consumed. <u>During reverse execution, the Boolean predicate is</u>

[6]Extra credit problem for the reader: why is it called Ψ-Lisp?

applied to the value previously returned by the if-then-else expression to determine which arm to execute. In order to keep the programmer honest, this predicate is also applied to the value *to be returned*, to make sure that it returns true on the then arm, and false on the else arm. To allow for more traditional programming styles, we allow this boolean predicate to push some state on a hidden conditional "history" stack during forwards execution, which is popped during reverse execution to remember the direction of the branch. Since only a single bit of information is being saved for a conditional expression, and only by *some* conditional expressions, this bit stack can be implemented very efficiently. A clever optimizing compiler may be able to sometimes deduce the reverse predicate which doesn't use the bit stack; functions like these will operate with bounded history stacks and are analogous to tail recursion.

Here is a reversible version of the factorial function restricted to n>0 so as to be injective; the result is 1 if and only if the "then" arm was taken, which forces n=1.

```
(defun fact (n) (assert (and (integerp n) (> n 0)))
  (if (onep n) #'onep n (* n (fact (1- n)))))
```

Our interpretation so far is approximately consistent with an applicative/functional interpretation of the lambda-expression. We now give another, more operational, interpretation. Parameters are not passed by copy or by reference, but by "swap-in, swap-out" [Harms91]. When a variable is referenced as an argument in an expression, it is swapped out of the variable and into the argument list structure, so that a side-effect of computing the expression is to make all of the referenced variables unbound! This is why a variable should not appear more than once in an expression—succeeding occurrence will be unbound. Results are also returned by swapping, so that when a lambda-expression terminates, none of its local variables have any values.

Ψ-Lisp `EVAL` and `APPLY` look similar to their traditional Lisp counterparts, and their familiar appearance masks their metacircular ability to run backwards. There are some notable differences, however. Since `EVAL` must be reversible given only its computed value(s), `EVAL` returns 3 values: the expression, the environment and the computed value. `APPLY` is more symmetrical—it is given a function and an argument list, and returns the function and its value(s). Since many functions return multiple values, we make `APPLY` completely symmetrical—it consumes an argument list and returns a value list, which must be destructured by its recipient. We utilize these additional returned values to eliminate the need for the interpreter itself to save state, although the interpreted program itself may do so.

The code for a recursive function is incrementally copied during a recursive evaluation in a Y-like manner. Unfortunately, all of these copies become available at the end of the recursion and must be recycled. A more "efficient" scheme might keep multiple copies of the code on a separate "free list", so that they wouldn't have to be created during recursion. Tail recursive functions can not only reuse their stack frame, but their code, as well! Linear Lisp [Baker92LL] utilizes a more efficient scheme (similar to "copy on write") for managing copies.

Time and Space Complexity for a Reversible Garbage Collector

If we make the reasonable assumption (for a machine with a single level RAM memory) that the inverse execution of each instruction takes exactly the same amount of time as its forward execution, then the total time required is exactly double that of the non-collected computation, plus whatever time is required to copy the answer.

If the entire computation finishes before being reversed, then "garbage collection" has not helped at all. One can reverse portions of the computation at finer scales, however, and achieve significant storage savings. In this case, the "answer" which must be copied before the reversal is initiated is the summary of the computation "so far". In other words, we must "save" this intermediate state for later mutation (and reversal) at a larger scale. The saving of this intermediate state is equivalent to traditional marking!

The value returned from a subprogram is a good example. The "answer" can be copied, and a reversal initiated to collect any excess storage that the subprogram used, before the caller resumes "forward" motion again. This reversal is equivalent to the popping of a classical stack frame. In fact, our "collection is reverse mutation" hypothesis can simulate most standard garbage collection optimizations.

Reverse Execution Paging

Researchers have found that garbage collection algorithms have much less reference locality than "normal" programs. Garbage collectors therefore page extensively, and many garbage collection improvements are aimed more at reducing the paging/caching costs than in reducing the number of instructions executed.

If we perform a "full" garbage collection, then every node and every link must be traced, requiring that every page/cache line be visited even if it has not otherwise been touched or modified since the last garbage collection. If we perform a "partial" garbage collection in a generational system, then optimizations exist to avoid visiting otherwise untouched or unmodified pages. On the other hand, even a generational garbage collector will have to spend some time tracing in the pages recently referenced or modified by the mutator.

If one believes the "garbage collection by reverse mutation" hypothesis, then the collector must visit the same pages/cache lines as the mutator, but does not bother with reversing a lot of extraneous computation not involved with pointers. As a result, the collector will tend to page "more heavily" than the forward computation, simply because the reverse of the operations not causing page faults are irrelevant to the task of garbage collection! Thus, although it may not visit any more pages than the forward computation, the collector appears to be less efficient, because it has less to do on each page.

If we know that collection is the reverse of mutation, we may be able to use this information to greatly improve the paging of the collector. We first note that the last-in, first-out (LIFO) behavior of a stack works extremely well with the least-recently-used (LRU) page replacement algorithm; even very small stack caches have high hit rates with LRU. It is also well known that the optimal paging algorithm (OPT) can be computed by submitting the reversed reference stream to an LRU algorithm. But we have already submitted our mutator program to the LRU algorithm during forwards execution, so we can compute the information necessary to utilize the OPT paging algorithm during the reversed execution (collection) phase!

Thus, while our garbage collection by reversed mutation would seem to exactly double the execution time of the mutator, we might reduce the collection time in the context of a paged implementation by utilizing the additional information generated during forward execution, and thereby achieve a running time only a fraction longer than an uncollected computation.[7]

Implications for Real Hardware

The biggest problem with an exchange-oriented architecture is the fact that it goes squarely against the grain of one of the most universally-held assumptions about computation—that copying by reference is free. Conversely, exchanges are cheaper than copies, even though "everyone knows" that exchanges take 3 copies, and that exchanges on a bus require atomic (i.e., slow) back-to-back bus cycles.

That exchanges are expensive seems to be an artifact of traditional Boolean—i.e., irreversible—logic. One can conceive of other types of logic in which the same connection could be used for signalling in both directions simultaneously—e.g., optical fiber. In fact, since all suitably small physical systems are reversible, it is actually more difficult to build irreversible/non-exchange architectures out of small components than to build reversible/exchange architectures.

Interestingly enough, one can find a precedent for exchange architecture in current *cache consistency* protocols for MIMD multiprocessors. In order to cut down on the bus/network traffic in the case that multiple writers are active, some cache line may be *owned* by a particular processor. If another processor attempts to write to this line, then the ownership of the line may be transferred to the other processor, in which case the first processor no longer owns it. While the information going in the direction opposite to that of moving cache line is essentially a "hole", it is not difficult to conceive of a more productive exchange protocol which would allow them to immediately synchronize, for example.

The restricted fan-in/fan-out of exchange architectures provides relief to the designer of a PRAM implementation, because the unlimited fan-in/fan-out of a CRCW PRAM architecture [Corman90] is very expensive to achieve.

What Makes the Free-List Free?

Joseph Halpern, *et al*. [Halpern84], asked the question "what makes the free-list free?" We believe that their paper could not properly answer the question, because it is a *thermodynamic* question, not a *semantic* question. Our answer is that the free-list is free because it is storage that is in the highest state of availability; in thermodynamic terms, it is *work*. What is work? Carnot's theorem tells us that work (available energy) can be produced from heat (energy+randomness), by removing the randomness into a cooler reservoir. The free-list always has exactly the same configuration—a semi-infinite sequence of cells (list of nil's)—i.e., the structure of the free-list is isomorphic to ω, the first infinite ordinal. Since $1+\omega=\omega$, allocating a cell from the free-list does not change its structure. Since the structure of the free-list is always the same, it has *zero* entropy/information. No other simple structure for the free-list would have as little entropy/information.

[7]For example, the TI Explorer GC "learns" locality by letting the running program copy incrementally; this scheme seems to provide locality superior to that from any uniform (depth-first or breadth-first) copying strategy.

Conclusions and Previous Work

We have advocated the use of reversible models of computation, and we have shown how a particular model can be programmed to perform interesting computations. The property of reversibility is very strong, and should allow static analysis of programs to produce deeper information than is typically feasible with non-reversible models.

Hoare's, Dijkstra's and Pratt's logics all consider the operation of a program in reverse, although to our amazement, no one seems to have taken up the retrospectively obvious line of attack we here propose.[8] *Abstract interpretation* and *type inference* often consider the retrograde execution of a program. Computer architects find that the reverse of reference strings are an important concept in the study of memory hierarchies; e.g., the optimal page replacement algorithm is the least-recently-used algorithm running on the reversed reference string.

Our Ψ-Lisp bears much resemblance to dataflow architectures. Its argument-consuming property neatly finesses the storage recovery problems addressed by [Inoue88].

The Deutsch-Schorre-Waite "pointer-reversing" list tracing algorithm [Knuth73] is a paradigm of reference-count-conserving programming. Binding tree "re-rooting" [Baker78b] is an even more sophisticated reference-count-conserving algorithm. Suzuki [Suzuki82] gives the fundamental properties of pointer exchange and rotate instructions; Common Lisp's `rotatef` operation comes directly from this paper.

The `unwind-protect` and `wind-unwind` operations of Common Lisp and Scheme, respectively, offer a form of "undoing" computation. In fact, some implementations of `wind-unwind` utilize the reversible "state space" tree re-rooting [Baker78SB].

The reverse execution abilities of *Prolog* were initially touted, but never developed, and modern Prologs cannot run predicates backwards (this is not the same as backtracking!). Subjunctive computing has been proposed as a valuable programming style [Zelkowitz73] [Nylin76] [Lafora84] [Heering85] [Leeman85] [Leeman86] [Strothotte87]. An enormous literature has developed around reversibility for debugging [Balzer69] [Grishman70] [Archer84] [LeBlanc87] [Feldman88] [Pan88] [Wilson89] [Tolmach90] [Agrawal91].

Hofstadter devotes a portion of his Pulitzer prize-winning book [Hafstadter79] to palindromes and "crab canons", which are musical pieces that simultaneously have same theme going in both the forward and reverse directions. One can conceive of the mutator process as playing a theme, while the collector process plays the retrograde theme.

Bill Gosper was an early investigator [Beeler72] into the power of the Digital Equipment Corporation's PDP-10 `EXCH` instruction, which we have appropriated for our reversible computer. Swapping has been given new life as a fundamental synchronization primitive in shared-memory multiprocessor architectures in the form of the "compare-and-swap" operation, which Herlihy has proven to be powerful and universal [Herlihy91].

Acknowledgements

We appreciate the discussions with Peter Deutsch, Richard Fateman, Richard Fujimoto, Bill Gosper, Robert Keller, Nori Suzuki, Tommaso Toffoli, and David Wise about these concepts.

References

Abadi, M. & Plotkin, G.D. "A Logical View of Composition and Refinement". *Proc. ACM POPL 18* (Jan. 1991),323-332.

Agrawal, H. *et al*. "An Execution-Backtracking Approach to Debugging". *IEEE Software 8*,3 (May 1991),21-26.

Appel, A.W. "Simple Generational Garbage Collection and Fast Allocation". *Soft. Prac. & Exper. 19*,2 (Feb. 1989), 171-183.

Appel, A.W. "A Runtime System". *Lisp & Symbolic Comput. 3*,4 (Nov. 1990),343-380.

Archer, J.E., *et al*. "User recovery and reversal in interactive systems". *ACM TOPLAS 6*,1 (Jan. 1984),1-19.

Bacon, David F., *et al*. "Optimistic Parallelization of Communicating Sequential Processes". *Proc. 3rd ACM Sigplan PPOPP*, Williamsburg, VA, April, 1991,155-166.

Baker, H.G. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978),565-569.

Baker, H.G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Progr.*, June 1990,218-226.

Baker, H.G. "The Nimble Type Inferencer for Common Lisp-84". Tech. Report, Nimble Computer, 1990.

Baker, H.G. "CONS Should not CONS its Arguments, or, A Lazy Alloc is a Smart Alloc". *ACM Sigplan Not. 27*,3 (March 1992),24-34.

Baker, H.G. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

---

[8]Linearity, even without reversibility, elegantly eliminates the *aliasing* problem which gives these logics fits!

Baker, H.G. "Cache-Conscious Copying Collectors". *OOPSLA'91 GC Workshop*, Oct. 1991.
Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Not. 27*,8 (Aug. 1992), 89-98.
Balzer, R.M. "EXDAMS: Extendable Debugging and Monitoring System". *Proc. AFIPS 1969 SJCC 34*, AFIPS Press, Montvale, NJ,567-580.
Barghouti, N.S. & Kaiser, G.E. "Concurrency Control in Advanced Database Applications". *ACM Comput. Surv. 23*,3 (Sept. 1991),269-317.
Barth, J. "Shifting garbage collection overhead to compile time". *CACM 20*,7 (July 1977),513-518.
Barth, Paul S., *et al.* "M-Structures: Extending a Parallel, Non-strict, Functional Language with State". *Proc. Funct. Progr. Langs. & Computer Arch.*, LNCS 523, Springer-Verlag, Aug. 1991,538-568.
Barton, Ed. Conservative Logic. 6.895 Term Paper, MIT, May, 1978.
Bawden, Alan. "Connection Graphs". *Proc. ACM Conf. Lisp & Funct. Progr.*, Camb., MA, Aug. 1986.
Beeler, M., Gosper, R.W, and Schroeppel, R. "HAKMEM". AI Memo 239, MIT AI Lab., Feb. 1972. Important items: 102, 103, 104, 149, 150, 161, 166, 172.
Benioff, Paul. "Quantum Mechanical Hamiltonian Models of Discrete Processes that Erase Their Own Histories: Application to Turing Machines". *Int'l. J. Theor. Phys. 21* (1982),177-201.
Bennett, Charles. "Logical Reversibility of Computation". *IBM J. Res. Develop. 6* (1973),525-532.
Bennett, Charles. "Thermodynamics of Computation". *Int'l. J. Theor. Phys. 21* (1982),905-940.
Bennett, Charles. "Notes on the History of Reversible Computation". *IBM J. Res. Develop. 32*,1 (1988),16-23.
Bennett, Charles. "Time/Space Trade-offs for Reversible Computation". *SIAM J. Computing 18*,4 (Aug. 1989).
Berry, G., and Boudol, G. "The Chemical Abstract Machine". *ACM POPL 17*, San Francisco, CA, Jan. 1990.
Chase, David. "Garbage Collection and Other Optimizations". PhD Thesis, Rice U., Nov. 1987.
Chen, W., and Udding, J.T. "Program Inversion: More than Fun!". *Sci. of Computer Progr. 15* (1990),1-13.
Chen, W. "A formal approach to program inversion". *Proc. ACM 18th Comp. Sci. Conf.*, Feb., 1990,398-403.
Cheney, C.J. "A Nonrecursive List Compacting Algorithm". *CACM 13*,11 (Nov. 1970),677-678.
Clarke, E.M. "Synthesis of resource invariants for concurrent programs". *ACM TOPLAS 2*,3 (July 1980).
Cohen, Jacques. "Non-Deterministic Algorithms". *Comput. Surveys 11*,2 (June 1979),79-94.
Corman, T.H., *et al*. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
Cousot, P., and Cousot, R. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". *Proc. ACM POPL 4* (1977),238-252.
Coveney, P.V. & Marcer, P.J. "Irreversibility and computation". *Specs. in Sci. & Tech. 14*,1 (1991?),51-55.
DeWitt, B. & Graham, N., *eds*. *The Many-Worlds Interpretation of Quantum Mechanics*. Princeton U. Press, 1973.
Deutsch, D. "Quantum Theory, the Church-Turing Hypothesis, and Universal Quantum Computers". *Proc. Roy. Soc.* (1985).
Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
Dobkin, D.P., and Munro, J.I. "Efficient Uses of the Past". *Proc. ACM FOCS 21* (1980),200-206.
Drescher, G.L. "Demystifying Quantum Mechanics: A Simple Universe with Quantum Uncertainty". *Complex Sys. 5* (1991),207-237.
Feldman, S., and Brown, C. "IGOR: A System for Program Debugging via Reversible Execution". *Proc. Sigplan/Sigops WS on Parl & Distr. Debugging*, May 1988,112-123.
Feynman, Richard P., *et al*. *The Feynman Lectures on Physics, Vol. I.* Addison-Wesley, Reading, MA, 1963.
Feynman, Richard P. "Quantum Mechanical Computers". *Founds. of Physics 16*,6 (1986),507-531.
Fisher, J. "Trace scheduling: A technique for global microcode compaction". *IEEE Tr.. Comps. C-30*,7 (July 1981),478-490.
Floyd, R.W. "Nondeterministic Algorithms". *J. ACM 14*,4 (Oct. 1967),636-644.
Fredkin, E., and Toffoli, T. "Conservative Logic". *Int'l. J. Theor. Physics 21*,3/4 (1982),219-253.
Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.
Grishman, R. "The debugging system AIDS". *AFIPS 1970 SJCC 41*, AFIPS Press, Montvale, NJ 1193-1202.
Halpern, J.Y., *et al*. "The Semantics of Local Storage, or What Makes the Free-List Free?". *ACM POPL 11*, 1984,245-257.
Harel, David. *First Order Dynamic Logic*. Springer-Verlag LNCS 68, 1979.
Harms, D.E., and Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Trans. SW Engrg. 17*,5 (May 1991),424-435.
Harrison, P.G. "Function Inversion". In Jones, N., *et al*, *eds*. *Proc. Workshop on Partial Evaluation and Mixed Computation*, Gammel Avernaes, Denmark, Oct. 1987, North-Holland, 1988.
Hederman, Lucy. "Compile Time Garbage Collection". MS Thesis, Rice U. Comp. Sci. Dept., Sept. 1988.
Heering, J., and Klint, P. "Towards monolingual programming environments". *ACM TOPLAS 7*,2 (April 1985),183-213.
Herlihy, Maurice. "Wait-Free Synchronization". *ACM TOPLAS 11*,1 (Jan. 1991),124-149.
Hofstadter, Douglas R. *Gödel, Escher, Bach: an Eternal Golden Braid.* Vintage Bks., Random House, NY, 1979.

Inoue, K., *et al*. "Analysis of functional programs to detect run-time garbage cells". *ACM TOPLAS 10*,4 (Oct. 1988),555-578.

Johnsson, T. "Lambda lifting: transforming programs to recursive equations". *Proc. FPCA*, Nancy, France, Springer LNCS 201, 1985,190-203.

Kay, A.C. "Microelectronics and the Personal Computer". *Sci. Amer. 237*,3 (Sept. 1977),230-244.

Keller, Robert M., *et al*. "An Architecture for a Loosely-Coupled Parallel Processor". Tech. Rep. UUCS-78-105, Oct. 1978,50p.

Kieburtz, Richard B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976),95-107.

Kieburtz, R. B. "The G-machine: a fast, graph-reduction evaluator". *Proc. IFIP FPCA*, Nancy, France, 1985.

Kieburtz, Richard B. "A RISC Architecture for Symbolic Computation". *Proc. ASPLOS II, Sigplan Not. 22*,10 (Oct. 1987),146-155.

Korth, H.F., *et al*. "Formal approach to recovery by compensating transactions". *Proc. 16th Int'l. Conf. on Very Large Databases*, 1990.

Kung, H.T. & Robinson, J.T. "On optimistic methods for concurrency control". *ACM Trans. on DB Sys. 6*,2 (June 1981).

Jefferson, David R. "Virtual Time". *ACM TOPLAS 7*,3 (July 1985),404-425.

Lafont, Yves. "The Linear Abstract Machine". *Theor. Computer Sci. 59* (1988),157-180.

Lafont, Yves. "Interaction Nets". *ACM POPL 17*, San Franciso, CA, Jan. 1990,95-108.

Lafont, Yves. "The Paradigm of Interaction (Short Version)". Unpubl. manuscript, July 12, 1991, 18p.

Lafora, F, & Soffa, M.L. "Reverse Execution in a Generalized Control Regime". *Comp. Lang. 9*,3/4 (1984), 183-192.

Landauer, R. "Dissipation and Noise Immunity in Computation and Communication". *Nature 335* (Oct. 1988),779-784.

LeBlanc, T.J., and Mellor-Crummey, J.M. "Debugging parallel programs with Instant Replay". *IEEE Tr. Comp. 36*,4 (April 1987),471-482.

Leeman, G.B. "Building undo/redo operations into the C language". *Proc. IEEE 15th Annual Int'l. Symp. on Fault-Tolerant Computing*, 1985,410-415.

Leeman, G.B. "A Formal Approach to Undo Operations in Programming Languages". *ACM TOPLAS 8*,1 (Jan. 1986),50-87.

Levy, E., *et al.* "An Optimistic Commit Protocol for Distributed Transaction Management". *Proc. ACM SIGMOD*, Denver, CO, May 1991,88-97.

Lewis, H.R., & Papadimitriou, C.H. "Symmetric Space-bounded Computation". *Theor. Comp. Sci. 19* (1982),161-187.

Lieberman, H., & Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*, 6 (June 1983),419-429.

Lindstrom, Gary. "Efficiency in Nondeterministic Control through Non-Forgetful Backtracking". Tech. Rep. UUCS-77-114, Oct. 1977,18p.

MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.

Manthey, M.J., & Moret, B.M.E. "The Computational Metaphor and Quantum Physics". *CACM 26*,2 (Feb. 1983),137-145.

Margolus, Norman. "Physics-Like Models of Computation". Elsevier North-Holland, *Physica 10D* (1984),81-95.

Margolus, Norman H. *Physics and Computation*. Ph.D. Thesis, MIT/LCS/TR-415, March 1988,188p.

Mattson, R.L., *et al*. "Evaluation Techniques for Storage Hierarchies". *IBM Sys. J. 9*,2 (1970),78-117.

McCarthy, John. "The Inversion of Functions defined by Turing Machines". In Shannon, C.E., and McCarthy, J., *eds*. *Automata Studies*, Princeton, 1956,177-181.

McDowell, C.E. & Helmbold, D.P. "Debugging concurrent programs". *ACM Comput. Surv. 21*,4 (Dec. 1989),593-622.

Miller, B.P. & Choi, J.-D. "A mechanism for efficient debugging of parallel programs". *Proc. ACM PLDI,* 1988,135-144.

Morita, K. "A Simple Construction Method of a Reversible Finite Automaton out of Fredkin Gates, and its Related Problem". *Trans. IEICE E 73*, 6 (1990),978-984.

Nylin, W.C.Jr., and Harvill, J.B. "Multiple Tense Computer Programming". *Sigplan Not. 11*,12 (Dec. 1976),74-93.

Pan, D.Z., and Linton, M.A. "Supporting reverse execution of parallel programs". *Proc. ACM Sigplan/Sigops WS on Par. & Distr. Debugging*, May 1988,112-123.

Penrose, R. *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*. Penguin Bks, London, 1989.

Peyton-Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice-Hall, NY, 1987.

Planck, Max. *Treatise on Thermodynamics*. Transl. Ogg, A., Dover Publ., NY, 1945.

Ressler, A.L. *The Design of a Conservative Logic Computer and a Graphical Editor Simulator*. M.S. Th., MIT, 1981, 128p.
de Roever, Willem P. "On Backtracking and Greatest Fixpoints". In Neuhold, Erich J., *Ed. Formal Description of Programming Concepts*, North-Holland, Amsterdam, 1978.
Romanenko, Alexander. "Inversion and Metacomputation". *ACM PEPM'91*, New Haven, CT, June 1991,12-22.
Rosenschein, Stanley J. "Plan Synthesis: A Logical Perspective". *Proc. IJCAI-81*, Vancouver, Canada, Aug. 1981, 331-337.
Ruggieri, C. & Murtagh, T. P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*,285-293.
Schorr, H., & Waite, W.M. "An efficient machine-independent procedure for garbage collection in various list structures". *CACM 10*,8 (Aug. 1967),501-506.
Shoman, Y., and McDermott, D.V. "Directed Relations and Inversion of Prolog Programs". *Proc. Conf. of 5th Gen. Comp. Sys.*, ICOT, 1984.
Sleator, D.D. & Tarjan, R.E. "Amortized Efficiency of List Update and Paging Rules". *CACM 28*,2 (Feb. 1985),202-208.
Smith, J.M., and Maguire, G.Q., Jr. "Transparent concurrent execution of mutually exclusive alternatives". *Proc. 9th Int'l. Conf. on Distr. Computer Sys.*, Newport Bch., CA, June 1989.
Strom, R.E., *et al*. "A recoverable object store". IBM Watson Research Ctr., 1988.
Strothotte, T.W., and Cormack, G.V. "Structured Program Lookahead". *Comput. Lang. 12*,2 (1987),95-108.
Suzuki, N. "Analysis of Pointer 'Rotation'". *CACM 25*,5 (May 1982)330-335.
Toffoli, T. "Reversible Computing". MIT/LCS/TM-151, Feb. 1980, 36p.
Toffoli, T. "Reversible Computing". In De Bakker & van Leeuwen, *eds*. *Automata, Languages and Programming,* Springer-Verlag (1980),632-644.
Toffoli, T. "Bicontinuous Extensions of Invertible Combinatorial Functions". *Math. Sys. Theor. 14* (1981),13-23.
Toffoli, T. "Physics and Computation". *Int'l. J. Theor. Phys. 21*, 3/4 (1982),165-175.
Tolmach, A.P., and Appel, A.W. "Debugging Standard ML without Reverse Engineering". *Proc. ACM Lisp & Funct. Progr. Conf.*, Nice, France, June 1990,1-12.
Turner, D. "A New Implementation Technique for Applicative Languages". *SW—Pract.&Exper. 9* (1979),31-49.
Vitter, J.S. "US&R: a new framework for redoing". *ACM Symp. on Pract. SW Dev. Envs.*, Pitts., PA, April 1984,168-176.
Wadler, P. "Views: A way for pattern matching to cohabit with data abstraction". *ACM POPL 14* (1987),307-313.
Wadler, P. "Is there a use for linear logic?". *Proc. ACM PEPM'91*, New Haven, June, 1991,255-273.
Wakeling, D. & Runciman, C. "Linearity and Laziness". *Proc. Funct. Progr. & Comp. Arch.*, Springer LNCS 523, 1991,215-240.
Wilson, P.R. & Moher, T.G. "Demonic memory for process histories". *Proc. Sigplan PLDI*, June 1989.
Zelkowitz, M.V. "Reversible Execution". *CACM 16*,9 (Sept. 1973),566-566.
Zurek, W.H., *ed*. *Complexity, Entropy and the Physics of Information*. Addison-Wesley, Redwood City, 1990.