Pure versus Impure Lisp

Nicholas Pippenger*
(nicholas@cs.ubc.ca)

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1Z4
CANADA

Abstract: The aspect of purity versus impurity that we address involves the absence versus presence of mutation: the use of primitives (`RPLACA` and `RPLACD` in Lisp, `set-car!` and `set-cdr!` in Scheme) that change the state of pairs without creating new pairs. It is well known that cyclic list structures can be created by impure programs, but not by pure ones. In this sense, impure Lisp is "more powerful" than pure Lisp. If the inputs and outputs of programs are restricted to be sequences of atomic symbols, however, this difference in computability disappears. We shall show that if the temporal sequence of input and output operations must be maintained (that is, if computations must be "on-line"), then a difference in complexity remains: for a pure program to do what an impure program does in n steps, $O(n \log n)$ steps are sufficient, and in some cases $\Omega(n \log n)$ steps are necessary.

* This research was partially supported by an NSERC Operating Grant.

1. Introduction

The programming language Lisp (see McCarthy [7] and McCarthy *et al.* [8]) was inspired by the λ-calculus (see Church [2]), and most of its basic features are frank imitations of aspects of the λ-calculus (with the most essential differences being in the rules for order of evaluation). In this way Lisp became the first significant programming language to allow the computation of all partial recursive functions by purely applicative—or functional—programs, without side-effects.

One feature of Lisp that goes beyond the applicative realm is the inclusion of primitives for what is now usually called "mutation". These primitive have semantics rooted in the von Neumann architecture of the machines on which Lisp was implemented, rather than in the λ-calculus. The main primitives for mutation in Lisp are `RPLACA` and `RPLACD`, which mutate the components of an existing pair (in contrast with `CONS`, which creates a new pair). We shall refer to Lisp with or without these mutation primitives as *pure* or *impure* Lisp, respectively. (This usage is fairly common, but it must be admitted that these terms are often used with reference to other features of programming languages—indeed, for any features that happen not to fit conveniently within the writer's conceptual framework.)

Our goal in this paper is to assess the extent to which mutation primitives add something essential to the language, and the extent to which they can be simulated—or eliminated in favor of—the purely applicative primitives of the language. As is often the case, the question can be formulated in several ways; our immediate goal is to describe the formulation we have in mind, and to explain why we have chosen it in preference to others.

We begin with a trivial observation. If pairs, once created, can never be mutated, then their components can only be references to previously existing objects: all the arrows in box-and-arrow diagrams point backward in time, and thus these diagrams are acyclic. It follows that if we allow the outputs of programs to be the data structures represented by such diagrams, then `RPLACA` and `RPLACD` do indeed add something essential, for they make possible the creation of structures whose diagrams contain cycles. This answer is not completely satisfying, however, because it assumes we want our programs to produce a particular representation of the answer, and it is the representation—rather than the answer—that is beyond the power of pure Lisp.

When we redirect our attention from representations of answers to the answers themselves, we are led to the observation that, when the inputs and outputs of programs are words over a finite alphabet (or, alternatively, natural numbers) both pure and impure Lisp compute all and only the partial recursive functions, and thus are equivalent in power.

This answer is also not completely satisfying: while it describes what computations can be performed, it ignores the issues of the efficiency of these computations.

When we redirect our attention from computability theory to complexity theory we obtain a crisp formulation of our problem. Consider computational problems that have as input a word over a finite alphabet (say a sequence of Boolean values) and have as output a yes/no answer (another Boolean value). Can every impure Lisp program solving such a problem be transformed into a pure Lisp program with the same input/output behavior, in such a way that number of primitives executed by the pure program exceeds the number performed by the impure program by at most a constant factor? Unfortunately, we are unable to answer this question. Only after putting two additional restrictions on computations will we be able to delineate precisely the additional computational power conferred by the impure primitives.

We shall say that a computation is *symbolic* if its input and output each consist of sequences of atomic symbols. These symbols can be incorporated as the components of pairs, and can be distinguished from pairs by a predicate `ATOM`, but the only other operation that can be performed on them is the two-place predicate `EQ`, which tests for equality of atomic symbols. The crucial property of atomic symbols is that there is an unlimited supply of distinct symbols, so that a single symbol can carry an unbounded number of bits of information. The fact that equality is the only predicate defined on symbols makes it very inefficient to convert the information they carry into any other form. Nevertheless, atomic symbols are a very natural part of the Lisp world-view, and insisting that they be treated as such seems less artificial that allowing primitives (such as the `EXPLODE` and `IMPLODE` in some dialects of Lisp) that allow them to be treated as composites.

We shall say that a computation is *on-line* if its input and output each comprise an unbounded sequence of symbols and if, for every n, the n-th output is produced by the computation before the $(n+1)$-st input is received. This notion refers to an unending computation, and some convention is necessary to reconcile it with the customary view of Lisp programs as functions with finitely many arguments and a single value. We shall regard on-line computations as being performed by non-terminating programs with no arguments, which receive their inputs using a primitive `READ` operation and produce their outputs using a primitive `WRITE` operation. These new primitives have side-effects, of course; it should be borne in mind that "purity" refers to the absence of `RPLACA` and `RPLACD` operations, rather than to an absence of side-effects. On-line computation is not part of the classical Lisp world-view, but it is a natural component of interactive transaction-processing systems.

With these notions we can now state our main result.

Theorem 1.1: There is a symbolic on-line computation that can be performed by an impure Lisp program in such a way that at most $O(n)$ primitive operations are needed to produce the first n outputs, but for which every pure Lisp program must perform at least $\Omega(n \log n)$ primitive operations (for some inputs sequences, and for infinitely many values of n) to produce the first n outputs.

(Here $\Omega\big(f(n)\big)$ represents a function of n bounded below by a positive constant multiple of $f(n)$.) That this result is the best possible, to within constant factors, is shown by our second result.

Theorem 1.2: Every symbolic on-line computation that can be performed by an impure Lisp program in such a way that at most $T(n)$ primitive operations are needed to produce the first n outputs, can be performed by a pure Lisp program that performs at most $O\big(T(n) \log T(n)\big)$ primitive operations (for all inputs sequences, and for all values of n) to produce the first n outputs.

To the objection that the efficiency of mutation is too well known or obvious to warrant proof, we offer the following argument. It is well known that a last-in-first-out stack discipline is easily implemented in pure Lisp, but the obvious implementation of a first-in-first-out queue relies on mutation to add items at the tail of the queue. But Fischer, Meyer and Rosenberg [3] have shown by an ingenious construction that a queue (or even a dequeue, where items can be added or removed from either end) can be implemented in pure Lisp with $O(1)$ primitive Lisp operations being performed for each queue (or dequeue) operation. In the face of this highly non-obvious implementation, it is unconvincing to claim without proof that there is not an even more ingenious and non-obvious implementation of a full interpreter for impure Lisp in pure Lisp, with $O(1)$ primitive pure Lisp operations being performed for each primitive impure Lisp operation. Indeed, after rediscovering a special case of the Fischer, Meyer and Rosenberg result, Hood and Melville [5] conclude: "It would be interesting to exhibit a problem for which the lower bound in Pure LISP is worse that some implementation using **rplaca** and **rplacd**."

2. Discussion

The question we address seems implicit in much of the Lisp literature, but the first explicit formulation we have found is due to Ben-Amram and Galil [1], who mention the cyclic/acyclic distinction, then go on to ask about the complexity of simulation. The use of input values to which only certain restricted operations can be applied is well established in the comparison of programming languages according to "schematology", as introduced by Paterson and Hewitt [9]. The first use of schematology for a comparison based on complexity rather than computability is due to Pippenger [10]. The special case of atomic symbols, in the sense used here, was considered by Tarjan [13]. The restriction to on-line computation is well established in the literature of automata theory; see Hennie [4].

We should say a few words about the models we use to embody the powers of pure and impure Lisp. These models will be the pure and impure Lisp machines. Such a machine will be furnished with a built-in program which takes the form of a flowchart, with recursion being implemented by explicit manipulation of a pushdown stack. Specifically, we consider programs that manipulate values (which may be atomic symbols or pairs) in a fixed number of registers; these registers contain mutable values, even in the pure case. The primitive operations are the predicates `ATOM` and `EQ` (which appear in the decision lozenges of flowcharts) and the operations `READ`, `WRITE`, `CONS`, `CAR` and `CDR` (which also take their arguments from and deliver their values to registers) and, in the impure case, the mutation operations `RPLACA` and `RPLACD`.

The use of flowchart models allows us to ignore questions of variable binding and scope, since the primitives of even the pure model allow any of the common scoping conventions to be simulated efficiently. (We are assuming here that a particular program, including any attendant subprograms, involves just a fixed number of variable names, whose current bindings can be kept in a fixed number of registers. There is no consideration here of mechanisms such a `EVAL` that would allow symbols from the input to be used as variable names and bound to values.) Flowchart models also allow us to ignore questions of control structures: as mentioned above, recursion can be simulated by manipulation of a pushdown stack; many other control structures, such as explicit use of current continuations, could similarly be simulated.

We have not allowed for constants (such as `NIL`) to be incorporated into programs, or for other uses of `QUOTE`. Our justification for this is as follows. Any program will involve only a fixed number of constants, and their only use is to be compared (via `EQ`) to other atomic symbols. If k such constants are needed, we may assume that the input sequence begins by presenting them in some agreed upon order, and that they are to be echoed back as the first k outputs. The program should test that they are in fact pairwise distinct; if so it can then proceed with the original computation; if not it can substitute some agreed upon dummy computation (such as eternally echoing back inputs and outputs).

If we overlook the presence of atomic symbols, the impure model is very similar to the Storage Modification Machines introduced by Schönhage [12] (which in turn have have the model of Kolmogorov and Uspenskiĭ [6] as a precursor).

3. The Upper Bound

We reformulate Theorem 1.2 in terms of machines as follows.

Theorem 3.1: Every on-line symbolic impure Lisp machine M can be simulated by a pure Lisp machine M' in such a way that all outputs produced by M within the first n steps are produced by M' within the first $O(n \log n)$ steps.

This theorem is established by the construction of a trite simulation, which will not be given in detail here. It can be obtained by modification of arguments given by Ben-Amram and Galil [1], but it is just as easy to describe the construction from scratch.

The key idea is to represent the state of the store in the impure machine by a balanced tree. The construction of new pairs by `CONS` is accomplished by allocating new paths in the tree, and the allocator issues new paths in order of increasing length, so that the tree is kept balanced. The fetches from store implicit in `CAR` and `CDR` operations, as well as the updates implicit in `RPLACA` and `RPLACD` operations, are performed by following paths in the tree from the root to the nodes containing the relevant information and, in the case of `RPLACA` and `RPLACD`, in rebuilding a modified version of the path while backtracking to the root. We observe that the constant implicit in the O-notation is *independent* of the machine M.

The "path copying" technique just described was applied by Sarnak and Tarjan [11] to the implementation of "persistent" data structures, in which old versions of the data structure can always be copied and updated independently. One of the benefits of a pure programming style (in the sense used here) is that all data structures are automatically persistent.

4. The Lower Bound

In this section we shall concoct a computation that separates the power of pure and impure Lisp machines. The proof that producing the first n outputs can be accomplished with $O(n)$ operations in impure Lisp will be easy. The proof that pure Lisp requires $\Omega(n \log n)$ operations, which we shall just sketch here, is the heart of the result. At a superficial level, this proof is an information-theoretic counting argument analogous, for example, to the one used to show that $\Omega(n \log n)$ comparisons are needed to sort n items.

It is not at all obvious, however, how such an argument can distinguish between creation and mutation. The key to the argument is to bring about a situation in which certain information, which can be measured by a counting argument, can be retrieved by impure operations at a rate of $\Omega(\log n)$ bits/operation, but by pure operations only at a rate of $O(1)$ bits/operation.

Consider a set of s "records" $R_1, \ldots, R_s$, each of which comprises an atomic symbol together with two pointers that are used to link the records into two linear chains. The first chain, which we call the A-chain, will link the records in the order

$$A \longrightarrow R_1 \longrightarrow \cdots \longrightarrow R_s \longrightarrow \texttt{NIL}.$$

The second chain, which we call the B-chain, will link the records in the order

$$B \longrightarrow R_{\pi(1)} \longrightarrow \cdots \longrightarrow R_{\pi(s)} \longrightarrow \texttt{NIL}.$$

where π is a permutation on $\{1, \ldots, s\}$.

We shall now describe the computational problem by describing how an impure Lisp machine solves it. The problem consists of a *prolog* comprising s^2 steps, followed by an unbounded sequence of *phases*, each comprising $2s$ steps.

The prolog takes place as follows. After checking that it has received two distinct input symbols to use as Boolean values, the impure machine M reads a tally notation for s (as $s-1$ trues follows by a false), and constructs as it does so the s records linked in the A-chain. (The B-chain links and atomic symbols are left unspecified.) Then, for each r from 1 to s, M reads a tally notation for $\pi(r)$, and fills in as it does so the B-chain links. (The atomic symbols are still left unspecified.) The prolog finishes by reading enough additional inputs to bring the number of steps up to s^2. (This is done just to make the number of steps in the prolog independent of the permutation π.)

The remainder of the computation is divided into phases of $2s$ steps each. During the first s steps of each phase, M reads in s atomic symbols and stores them in the records in their order according to the A-chain. During the last s steps of each phase, M writes out these s atomic symbols from the records in their order according to the B-chain. We have not specified what symbols M writes out during the prolog, or during the first half of each phase; we shall stipulate that it echos the last symbol read at each such step. (The output does not depend on the symbols M reads during the second half of each phase.) It is clear that M can perform this computation using $O(1)$ primitive operations between each `READ` and `WRITE` operation.

We shall restrict our attention at this point to input sequences for which all the atomic symbols read and written in all of the phases are distinct. This will allow us to ascribe each symbol produced by a `WRITE` operation during a phase to a well defined `READ` operation earlier in that phase. We observe that for any value of s, there are just $s!$ possible permutations π that might be described during the prolog.

It remains to show that any pure Lisp machine M' requires at least $\Omega(s \log s)$ primitive operations in each phase, for most choices of π, say for all but $(s-1)!/2$ choices of π. Since M performs just $O(s^2)$ operations in the prolog, and just $O(s)$ operations in each phase, we can then obtain Theorem 1.1 by considering the first s phases, for then M will produce $O(s^2)$ outputs using $O(s^2)$ operations, while M' will require $\Omega(s^2 \log s)$ operations for all but at most $s(s-1)!/2 = s!/2$ choices of π.

Consider an interval $[a, b]$ of steps. Let us say that a set of input sequences is (a, b)-*coherent* for the pure Lisp machine M' if, from the processing of the a-th input x_a to the b-th output y_b, all test operations (that is `ATOM` and `EQ` operations) have the same outcome. The operation of M', restricted to such an interval of operations and to a set of coherent inputs, corresponds to that of a "straight-line program", in which a fixed sequence of the primitive operations `CAR`, `CDR`, `CONS`, `READ` and `WRITE` is performed.

Let us say that an input sequence is (a, b)-*psittacine* for M' if each of the outputs $y_a, \ldots, y_b$ is equal to one of the inputs $x_a, \ldots, x_b$.

Lemma 4.1: Let $\mathcal{C}$ be any (a, b)-coherent set of inputs for a pure Lisp machine M', and suppose that the inputs in $\mathcal{C}$ are (a, b)-psittacine for M'. Then there exists a map $h : \{a, \ldots, b\} \to \{a, \ldots, b\}$ such that, for every input sequence $x_1, x_2, \ldots$ in $\mathcal{C}$, M' produces the output $y_i = x_{h(i)}$ for $i \in \{a, \ldots, b\}$.

Sketch of Proof: For $i \in \{a, \ldots, b\}$, start with the `WRITE` operation that produces y_i and trace back through the computation to the `READ` operation that received corresponding input x_j such that $y_i = x_j$. For each input sequence, we have that j is a well defined value in $\{a, \ldots, b\}$, and we must show that j is the same value $h(i)$ for all input sequences in $\mathcal{C}$.

We trace back in the following way. The `WRITE` operation that produces y_i takes the output value from some register. We trace back to the operation that put this value into the register. If this operation was a `READ` operation, we are done. Otherwise, it was a `CONS`, `CAR` or `CDR` operation. In this case we trace back to the operation that put the relevant argument into a register. We continue until we reach the appropriate `READ` operation. (The process of tracing back must terminate with a `READ` operation, since we have disallowed the use of `QUOTE` to introduce constants.)

Since M' is a pure Lisp machine, any pairs involved in the process above must have been constructed during the interval $[a, b]$ (we are starting from an output in the interval $[a, b]$, following pointers that point backward in time, and ending with an input in the interval $[a, b]$). It follows that all the operations involved in the process take place in the interval $[a, b]$, during which M' executes a straight-line program. Consequently, the same input x_j is reached from the output y_i, for every input sequence in $\mathcal{C}$, which is what was to be shown. $\triangle$

It is worth observing that this lemma breaks down for impure Lisp machines for two reasons: (1) the value produced by a `CAR` or `CDR` operation might trace back to a `RPLACA` or `RPLACD` rather than a `CONS`, and (2) pointers do not necessarily point backward in time, so we cannot conclude that all relevant operations take place in the interval $[a, b]$.

Now suppose that during some phase, corresponding to an interval $[a, b]$, M' performs at most t test operations for each of at least $(s-1)!/2$ choices of π. The outcomes of these operations partition a set of $(s-1)!/2$ input sequences into at most 2^t (a, b)-coherent classes. During each phase, M' writes out only symbols read in during the same phase, so each of these classes is (a, b)-psittacine for M'. Thus, by Lemma 4.1, the outputs produced by M' are specified by one of at most 2^t maps. But the behavior of the impure Lisp machine M calls for at least $(s-1)!/2$ different maps, since there are $(s-1)!/2$ different permutations π. Thus we have $2^t \geq (s-1)!/2$ or, by taking logarithms, $t = \Omega(s \log s)$. (Since each of the s phases needs $\Omega(s \log s)$ comparisons for all but the easy permutations (where "easy" means that fewer comparisons are needed), and since there are at most $(s-1)!/2$ easy permutations for each phases and s phases, ther are at most $s(s-1)!/2$ permutations that are easy for some phase. Hence at least half of the permutations $(s!/2)$ are not easy during any phase. Hence at least one permutation is not easy during any phase.)

We can see at this juncture the roles played by our two special restrictions. The symbolic inputs and outputs allow each step of a phase to involve $\Omega(\log s)$ bits of information (since there are s distinct symbols in each phase), while allowing the impure machine to process these bits using $O(1)$ operations. Thus, the lower bound would break down if we required the input sequence to be over a finite rather than an infinite set of symbols (and the upper bound would break down if we charged logarithmically for pointer manipulation operations, reflecting their implementation using a finite set of symbols). The on-line assumption allows the basic arguments to be repeated in s disjoint phases; without this assumption a pure machine could read the inputs for s phases before writing all the outputs for these phases, all using $O(s^2)$ operations.

5. Conclusion

We have shown that mutation can reduce the complexity of computations, at least when the computations are required to be performed on-line and when their inputs and outputs may be sequences of atomic values (rather than sequences of symbols drawn from a finite alphabet). We have further shown that this reduction is sometimes by as much as a logarithmic factor, but can never exceed a logarithmic factor.

Naturally it would be of interest to lift either or both of the special assumptions we have made. We would conjecture that a reduction in complexity can occur even for off-line computations and even for computations in which the inputs and outputs are words over a finite alphabet. Such a result, however, seems far beyond the reach of currently available methods in computational complexity theory.

6. References

[1] A. M. Ben-Amram and Z. Galil, "On Pointers versus Addresses", *J. ACM*, 39 (1992) 617–648.

[2] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, NJ, 1941.

[3] P. C. Fischer, A. R. Meyer and A. L. Rosenberg, "Real-Time Simulation of Multihead Tape Units", *J. ACM*, 19 (1972) 590–607.

[4] F. C. Hennie, "On-Line Turing Machine Computations", *IEEE Trans. on Electron. Computers*, 15 (1966) 34–44.

[5] R. Hood and R. Melville, "Real Time Queue Operations in Pure LISP", *Info. Proc. Lett.*, 13 (1981) 50–54.

[6] A. N. Kolmogorov and V. A. Uspenskiĭ, "On the Definition of an Algorithm", *AMS Translations (2)*, 29 (1963) 217–245.

[7] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Comm. ACM*, 3 (1960) 184–195.

[8] J. McCarthy *et al.*, *List* 1.5 *Programmer's Manual*, MIT Press, Cambridge. MA, 1962.

[9] M. S. Paterson and C. E. Hewitt, "Comparative Schematology", *Project MAC Conf. on Concurrent Systems and Parallel Computation*, (1970) 119–128.

[10] N. Pippenger, "Comparative Schematology and Pebbling with Auxiliary Pushdowns", *ACM Symp. on Theory of Computing*, 12 (1980) 351–356.

[11] N. Sarnak and R. E. Tarjan, "Planar Point Location Using Persistent Search Trees", *Comm. ACM*, 29 (1986) 669–679.

[12] A. Schönhage, "Storage Modification Machines", *SIAM J. Comput.*, 9 (1980) 490–508.

[13] R. E. Tarjan, "A Class of Algorithms That Require Nonlinear Time to Maintain Disjoint Sets", *J. Comput. and Sys. Sci.*, 18 (1979) 110–127.