Notes on Pippenger's Comparison of Pure and Impure LISP

Amir M. Ben-Amram
DIKU

June 12, 1996

On May 24th, I gave a talk in DIKU in which I described Nicholas Pippenger's new result on *pure versus impure LISP* (presented in POPL '96). This was followed by a discussion on the result and its implications. Some of the the ideas that were discussed are presented below.

1 Presentation of the Result

By *Pure LISP* we mean a first-order, strictly-evaluated language (contrast with lazy evaluation) that uses only `cons`, `car` and `cdr` to construct and examine data structures (starting with *atoms*. In its standard form, it is a pure-functional language in which functions do not have side effects and precisely correspond to mathematically-defined functions from the set of trees over atoms into the same set. The control-structure of the original language is also purely functional, meaning that a program is just a set of mutually-recursive functions whose bodies are expressions: there are no "commands," loops etc. However changing to an imperative control structure, in which there is only one program unit (no functions), that uses assignments, if's and loops or even goto's, does not make an difference for either expressive power or complexity analysis (this is simple to see because the language is strictly-evaluated and first-order).

By *Impure LISP* we mean the same language with the additional operators `setcar`, `setcdr` (or `rplaca`, `rplacd`). These operators changes parts of existing structures, giving rise to *mutation* (or *destructive update*).

It is not hard to show that any impure-LISP program running in time t can be compiled into a pure-LISP program running in time $O(t \log t)$: first implement the impure-LISP operations using an array of size at most t. Then represent the array as a balanced binary tree, which can be done in pure LISP.

The main result of the paper is a lower-bound theorem. It can roughly be described as follows. A problem P is presented, that can be solved in linear time, $t = O(n)$, in impure

1

LISP. It is proved that for any pure-LISP program p for P, the worst-case time complexity is $\Omega(n \log n)$.

2 Restrictions of the Proof and Open Problems

The lower-bound result requires two restrictive assumptions. We first describe the restrictions and their technical implications. Next, we discuss the two questions that should be asked regarding such assumptions: Are the restrictions necessary? Are they justifiable?

The first restriction is that the problem has to be solved *on-line*. This means, that after certain input is provided, an answer has to be output; the program is not allowed to read further input first. In contrast, an *off-line* program is one that reads all the input before delivering any output.

Note, that the purely-functional world view is only natural for off-line programs: the program computes a mathematically-defined function from lists (input sequences) to lists (output values). In fact, pure LISP does not allow a representation of an on-line program. This is one reason why Pippenger uses an imperative style and not a functional style (another could be that complexity is more transparent in this form). He extends the language with the imperative commands `read` and `write` for input and output.

The second assumption presumes that the input may include an *unbounded number of distinct symbolic atoms*. By *symbolic atoms* we mean atoms that are only amenable to equality tests. Standard LISP provides both symbolic atoms and numeric atoms.

Justification for Assumptions. As already said, symbolic atoms form a natural part of the LISP world-view. Another justification for making this assumption is that it coincides with the requirement that the program be *polymorphic* with respect to the type of input values. This is a requirement that could be made if the program is intended to operate as a module inside a larger software system.

I believe that the last suggestion could also justify the on-line requirement. Such a module may have to handle "messages" (in the object-oriented sense) to which it has to respond immediately — it is not acceptable to wait for more input before responding.

Necessity of Assumptions and Open Problems. Both assumptions are used in an essential way in the proof, so we know that at least this proof requires them. Moreover, the problem at hand could be solved in linear time by an off-line *pure* LISP program. This is an important observation to which we return below. It is therefore natural to pose the open problem:

- Does a similar lower bound hold for off-line problems?

As for the assumption of *unboundedly many symbolic atoms*, it was interesting to observe, that among the audience of my talk, functional programmers seemed quite ready to accept this assumption while non-functional programmers insisted that it was not realistic. For these programmers, the following questions are important:

- Does the lower bound hold if the atoms in the input are drawn from a fixed finite set?

- What if numeric atoms are allowed, on which the program may use arithmetics?

This may be the place to remark, that if the program may use arithmetics, but the input still consists of a series of *sympolic atoms* (so the language allows both types), Pippenger's proof applies. So in the last question, we assume that the input is numeric.

A last open problem that I find interesting concerns running time. Pippenger's proof involves a linear-time on-line program, i.e., a program which produces output at the same rate that it consumes input. It stores data into memory and uses them fast, making use of a fast-access data structure. However, consider programs whose running time is larger than linear. Here, one might hope that the (non-linear) time of constructing the data-structures needed for the simulation would be absorbed by the total running time. We thus ask

- Does a similar lower bound hold for impure-LISP programs of non-linear running time?

3 What About Cyclic Structures?

The most immediate observation on the power of destructive update operations in LISP is that they allow for the creation of *cyclic list structures*, impossible in pure LISP. Therefore, the question of *the power of destructive update* has been almost identified for many years with the question of *the power of cyclic structures*. It is striking, that the impure LISP program for solving Pippenger's problem *does not require cyclic structures*. Therefore, the latter question remains open! It can be formalized by looking at a language which has the following features: it has an imperative control structure, `read` and `write` commands, just like in Pippenger's presentation, except that assignments may have a new, recursive form:

```
Assignment ::= let Var := Exp
             | letrec Var₁ := Exp₁, ..., Varₙ := Expₙ
```

For example,

```
letrec X := cons 1 X
```

defines a circular structure, observationally equivalent to an infinite list of **1**'s. Note, that this observational equivalence does not hold for languages with destructive update, so our

extension is a "milder" way of creating circular structures. In fact it has most of the desirable features of the pure language, among them the very features that are used in Pippenger's lower-bound proof. Thus, this language, though it can create cyclic structures, falls on the *low side* of the gap shown by Pippenger's proof. Of course, it still seems to have an advantage over pure LISP.

Open problem. Can this advantage be demonstrated by a lower-bound proof?

Programming Research problem. Experiment with this language, to find out how many practical uses for circular structures can be realized in this "clean" form.

4 On-Line Programs and Lazy Evaluation

The above "letrec" construct is not common with strict functional languages, but it forms a standard part of *lazy* functional languages. Studying Pippenger's proof and its application in the last example, shows that selecting an evaluation strategy, whether strict or lazy, does not affect the validity of the proof. However, lazy languages have more to offer. They offer input-output in the form of *streams*. An input stream looks like a list, but elements of the list are only read from an input source when accessed by the program. This means, that a program can be written in a purely functional form, as a function that maps lists to lists, and still be an on-line program, because the input list will be read in gradually.

Recall that problem P from Pippenger's paper was solvable in linear time by pure LISP program in the *off-line* version. What if we take such an off-line program, and use a lazy language in which the input and output lists are changed to I/O streams?

If we are lucky, we may get a linear-time on-line program! "Lucky" here means that our program is designed so that it computes output values at the correct points in time with respect to the consumption of the input list. A program that has this property has been recently published [BJM], showing that the *gap between off-line and on-line pure LISP programs, or between "pure" and "impure" on-line programs, may sometimes be bridged by using lazy evaluation.*

Open problems.

- Show a gap between lazy and strict languages that does not depend on the use of I/O streams.

- Give a problem that allows the gap between "pure" and "impure" LISP to be demonstrated even for lazy languages with I/O streams.

4

We already know, that the latter result could not be obtained by the problem used in Pippenger's paper. The same is true for a larger class of similar problems. Recall that Pippenger's problem is solved by an impure program that has the following structure:
First, a certain data structure is built. Here destructive update is not essential.
Then, new data read on-line are repeatedly placed into the structure (here destructive update is used). The structure is used by the program for generating the desired output.

It is possible to generalize the strategy seen in [BJM] and obtain lazy programs (without destructive updates) that run efficiently on-line from any program that shares the above structure. I omit the details.

5   Recent Results

I have lately been able to show a certain gap between pure and impure LISP for *off-line* programs. The result is, briefly, that the *time-space* product $T_p(n) \cdot S_p(n)$ for a pure LISP program that solves the problem given by Pippenger (off-line) has to satisfy: $T_p(n) \cdot S_p(n) = \Omega(n^{3/2} \log n)$. Since the impure LISP program for this problem satisfies $T(n) \cdot S(n) = O(n^{3/2})$ we conclude, that a pure LISP program must take *either more time or more space* to solve the same off-line problem.

References

[BJM] R. Bird, G. Jones and O. de Moor, "A Lazy Pure Language versus Impure Lisp," http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications/FP-1-96.html