Sequence complexity

Axel Schumacher
Télécom ParisTech
Free Project

July 3, 2011

Contents

Acknowledgements

My warm thanks to Jon Harper, who helped me through the difficult process of learning Factor, and also to my two professors, Samuel Tardieu and Jean-Louis Dessalles, my supervisors, who spent time discussing my project and helping me.

1 Introduction

As a student at Télécom ParisTech, I chose to follow the 'Non-classical language paradigms' course with Samuel Tardieu[1], while working on a free project with Jean-Louis Dessalles[2]. Jean-Louis Dessalles suggested a project based on a Komolgorov complexity problem: given a small sequence of digits, shapes or characters, I had to design a small program able to complete this list as well as an eight-year-old child. [3]

The main goal of this project was to apply Kolmogorov theory to the given list in order to compress it in an optimal and human way. This will then enable us to extend that list using the compression. During the "Non-classical language paradigms" course, we studied uncommon languages such as Haskel, Scala or Factor; we also were to perform a small implementation project using one of those languages.

Factor[4] is a quite memory-close language, able to perform backtracking as well as Prolog using continuations and working on a stack. Its powerful and non-standard architecture makes it a very interesting language to work on, and this is why I chose to use it to implement my project.

2 Problem description

Human capabilities to compress and understand structures have always fascinated computer science researchers. Humans are able to perfectly perform tasks such as recognizing a person's face, understanding a language or, in our scope of interest, understanding and completing a list of digits, alphabetics or shapes. Those tasks are very difficult for machines, even using the most powerful computers. That is why much research is left to be done on the subject, always with the same goal: give machines more and more of our "human intelligence".

3 Basis

We will first illustrate the problem with an example. Let us imagine a very simple sequence:

$$1\ 2\ 2\ 3\ 3\ 3$$

Any eight-year-old child would easily suggest that a good continuation of this sequence would be:

$$1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 4$$

But how can one be so sure about it? This is where we can introduce the notion of "complexity", and more especially Kolmogorov complexity:

"In algorithmic information theory (a subfield of computer science), the Kolmogorov complexity of an object, such as a piece of text, is a measure of the computational resources needed to specify the object." [5]

In other words, this complexity has the particularity that it highly depends on the user, and is moreover really difficult to use inside a program.

Using this complexity, someone could define 1 2 2 3 3 3 as "count from one and repeat each number as much as itself", which is really easier to understand, memorize and extend for an human than a random sequence, like a telephone or credit card number. We will explore different ways to represent and manipulate such informations.

Another way to compute the complexity of a list would be to use *short term memory*, a list or tree containing the elements used last. An access to one of those elements would be its binary encoded address, for example. This model is easily explainable considering that a human, if he was able to solve a problem using a specific operator, would naturally try this operator first when confronted with a new problem. This list is implemented but not used because of its unpredictable nature.

That is why in the present version a standard *binary encoding* is used to compute the complexity of a list, with 0 costing 0.

Given a way to compute complexity, one can try to compress a list using *operators* while the resulting complexity is lower than that of the initial sequence.

4 Attempts

4.1 Three operators

At first, I thought about using Factor's *tuples*, special sequences of fields which are easily sharable between descendants, as in standard *Object Oriented Programming.*

I defined a tuple *operator* containing a field named *times.* Operator had the following methods, or *generic functions* in Factor:

- *cost*, a function returning an operator's cost which adds the values of its fields.
- *apply*, a function which uses a sequence and generates another sequence by uncompressing this operator on it (often using only its first element).
- *search*, a function which tries to find the current operator's application on a sequence.

Given those methods, I defined three operators:

- A copy operator: using its predecessor's *times* field, copies the given argument a *times* number of times.
- An increment operator: increments and copies the given argument a *times* number of times. If this argument is an operator, it will copy it by incrementing its *times* field.
- A step operator: also contains a *gap* argument and an *operator* field, applies its operator by skipping as many elements as given by *gap* on a list.

The step operator was useful in this case:

$$1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 4$$

When compressed once using copy operator, it gave:

$$(\textit{copy times}(1))\ 1\ (\textit{copy times}(2))\ 2\ (\textit{copy times}(3))\ 3\ (\textit{copy times}(4))\ 4$$

Here, we can easily see that we have an increment on the digits and on the operators, but separated by one element each time. Using the step operator, it gives:

$$(\textit{step operator times}(4)\ \textit{gap}(1)\ \textit{operator}(\textit{increment times}(1)))\ (\textit{copy times}(1))\ 1\ 2\ 3\ 4$$

The step operator will be applied four times with a gap of one. As between each gap we only have one element, the increment operator in *operator* field only applies once each time. The step operator will be applied to the directly following element, the copy operator. A final compression would give:

$$(\textit{step operator times}(4)\ \textit{gap}(1)\ \textit{operator}(\textit{increment times}(1)))(\textit{copy times}(1))$$

$$(\textit{increment times}(4))\ 1$$

In terms of complexity, if we count 1 for each operator plus the recursive cost of all its arguments, the raw sequence 1 2 2 3 3 3 4 4 4 4 will cost 23, while the compressed version costs only 14. Hence, we can say that we have a compressed version.

Copy and increment search were easy, I just had to compare the current result with the next sequence's element. However, for the step-operator, how could I determine the gap before searching? I tried each operator but with a gap of one. The problem was that it didn't use all the power of the step operator, but the search would cost a lot otherwise. And what would a step operator applied on a step operator mean? A very complex, not useful nor handy object actually.

Another problem is that it is difficult to extend a list: what specific operator should we "times-increment"? On this particular example, it is the first operator of the list, but can we be sure about it? Moreover, how can we be sure about the fact that a list *is* extensible?

This solution displeased Jean-Louis Dessalles, because of this not really "natural" strange step operator, and the difficulty for it to be understood and used. That is why he asked me to use only two operators: copy and increment. His point was that in the previous example, the increment operator should have been applied at the same time on copy operator and on digits. Therefore, I should put the argument into the operator instead of leaving it in the list. Then, the increment operator would have seen a sequence of copy operators incrementing at the same time their *times* and *argument* fields.

4.2 Two operators with an inelegant application

Now with this constraining instruction, the question was: "How can I apply different operators on the same object but on different fields?" I imagined a solution that I found ugly, but functional. Copy-operator would now know both its *argument* and its *times*. Increment-operator would inherit from copy-operator and moreover know where in its *argument* it would apply: on its *argument*, its *times* or both. This information was stored in a *where* field.

Here is an application on the previous example (the first set of parenthesis contains *argument*, the second set contains *times* and the last set contains *where*):

$$1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 4$$

$$(\mathit{copy}(1)(1))\ (\mathit{copy}(2)(2))\ (\mathit{copy}(3)(3))\ (\mathit{copy}(4)(4))$$

$$(\mathit{increment}(\mathit{copy}(1)(1))(4)(\mathit{both}))$$

The *where* field doesn't have any meaning when applied to a digit, and this is already a structural problem. This same field doesn't have any meaning either when applied to an increment-operator: it cannot apply to its *argument* because the contained increment-operator takes care of that, and hence can only apply on its *times*.

With this solution, some of the previous problems were solved: a list is only extensible if it can be reduced to one operator, and to extend it we only have to increment its *times* field and then recursively decompress it. Furthermore, putting any operator in the *argument* field of an increment-operator will always have a meaning, as we can imagine in the following more complicated situation:

$$1\ 1\ 1\ 2\ 1\ 1\ 2\ 1\ 2\ 3\ 1\ 1\ 2\ 1\ 2\ 3\ 1\ 2\ 3\ 4$$

which can be segmented in the following fashion:

$$1 \ast\ 1;\ 1\ 2 \ast\ 1;\ 1\ 2;\ 1\ 2\ 3 \ast\ 1;\ 1\ 2;\ 1\ 2\ 3;\ 1\ 2\ 3\ 4$$

Compression would give (the sequence has been cut for the sake of clarity):

$$(\mathit{increment}(1)(1)(\mathit{arg}))$$

$$(\mathit{increment}(1)(1)(\mathit{arg}))\ (\mathit{increment}(1)(2)(\mathit{arg}))$$

$$(\mathit{increment}(1)(1)(\mathit{arg}))\ (\mathit{increment}(1)(2)(\mathit{arg}))\ (\mathit{increment}(1)(3)(\mathit{arg}))$$

$$(\mathit{increment}(1)(1)(\mathit{arg}))\ (\mathit{increment}(1)(2)(\mathit{arg}))\ (\mathit{increment}(1)(3)(\mathit{arg}))\ (\mathit{increment}(1)(4)(\mathit{arg}))$$

Another time would give:

$$(\mathit{increment}(\mathit{increment}(1)(1)(\mathit{arg}))(1)(\mathit{times}))$$

$$(\mathit{increment}(\mathit{increment}(1)(1)(\mathit{arg}))(2)(\mathit{times}))$$

$$(\mathit{increment}(\mathit{increment}(1)(1)(\mathit{arg}))(3)(\mathit{times}))$$

$$(\mathit{increment}(\mathit{increment}(1)(1)(\mathit{arg}))(4)(\mathit{times}))$$

And a last one:

$$(\mathit{increment}(\mathit{increment}(\mathit{increment}(1)(1)(\mathit{arg}))(1)(\mathit{times}))(4)(\mathit{times}))$$

This solution is findable, apart from the complexity problem (see below), and the following problem: at the second iteration, how did it know that increment, when applied once, applied only to *times* and not to something else? I chose *times* because it has been able to compress at the last iteration, given that all the increment operators applied to *times*, but I had no right to do so.

During a meeting with both Samuel Tardieu and Jean-Louis Dessalles, this solution worked well enough, apart from some complexity problems: sometimes "good" (think "expected") solutions were not kept because they cost more than the uncompressed sequence. For example, Samuel Tardieu suggested 1 2 2 3 3 4, with segmentation 1 2 * 2 3 * 3 4 in mind, the program considered three increments in a row, each applied twice, but rejected it. For the "dirty" where-to-apply solution, Samuel Tardieu suggested that instead of a restrictive three-choice field, I should use an indexed list which could also handle deepness, indicating where the increment-operator should apply. This is the solution I adopted and customized in my final version.

5 Current solution

The current solution is based on Factor's suffix paradigm: an operator is represented by a letter, C for copy or I for increment, and applies to a specified number of arguments BEFORE it. Elements in a compressed sequence are only compressed sequences, digits or operator letters. My customized indexes work that way: increment operators own in their *where* field a sequence of digits, which are indexes representing an element's absolute position in the extended *argument*. For example, let us consider the following sequence:

$$\{\ a_0\ a_1\ \{\ a_2\ \{\ a_3\ a_4\ \}\ a_5\ \{\ a_6\ \}\ \}\ a_7\ \{\ a_8\ \}\ a_9\ \}$$

When extended, this sequence would be:

$$\{\ a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ \}$$

I used the extended indexes. Hence, any deepness level can be reached, and there is no need to recall structural information, as with tree indexes, in which case element a_4 would be accessed with 2.1.1 instead of simply 4. This notation is easier to manipulate and the resulting *where* sequence could even be compressed!

Jean-Louis Dessalles wasn't really satisfied with this index solution: it would not respect the Kantian structural form of the model (we should be able to manipulate shapes or vegetables as well, and use only them in the model). Moreover, the structure is absolutely not represented. He suggested the use of masks, more "natural" and compressed than my index notation, but this doesn't solve the Kantian problem and it is actually in bijection with my system (as it is more important to act simply and 'humanly' than to save RAM, we can neglect the compression difference).

For easier representation, an *argument* or a *where* field reduced to only one element is not contained in a sequence, but inserted as is.

For result purposes, C and I words won't cost anything.

We also define the order of the arguments of the operators:

- $WHAT\ TIMES\ C$
- $WHAT\ WHERE\ TIMES\ I$

Here is an example of the use of this system on our favorite example:

$$1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 4$$

$$1\ 1\ C\ 2\ 2\ C\ 3\ 3\ C\ 4\ 4\ C$$

$$\{\ 1\ 1\ C\ \}\ \{\ 0\ 1\ \}\ 4\ I$$

We only have one operator left, I, with just its arguments. We increment its *times* field:

$$\{\ 1\ 1\ C\ \}\ \{\ 0\ 1\ \}\ 5\ I$$

And decompress it to obtain its logical extension:

$$1\ 1\ C\ 2\ 2\ C\ 3\ 3\ C\ 4\ 4\ C\ 5\ 5\ C$$

$$1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 4\ 5\ 5\ 5\ 5\ 5$$

6 Backtracking with Factor

Let us now take a closer look at how the search algorithm works, using Factor's backtracking power.

6.1 amb and fail

Using continuations in factor, we can use the existing function amb to test all the elements of a sequence. When fail is called, the program backtracks to the last amb call and tries its next element. If there is no element left, this level is abandoned and the previous one is considered. This system, quite close to Prolog's, is useful for trying possibilities at different levels, and backtracks to the next possibility in case of failure.

6.2 Search order

My program is able to complete incomplete sequences: it will first try to remove a number of elements from 0 to the sequence length minus 2. At each removal, it will check if when extended the resulting sequence contains the initial one and fail otherwise.

If nothing is found, the program will extract the elements of the given sequence in such a way that all elements are present only once, and ordered from the last found to the first one. The program will then try to add all combinations, it starts with adding one element, then two, and so on until the length of the initial sequence is reached. This part has a theoretically factorial complexity, but in practice this is almost never reached because a solution is quickly found.

Given a sequence which we would like to search on, we try to compress:

At first, we try an operator, C and then I.

Given an operator, we decide a searching size. We will try from 1 up to the length of the sequence. A size represents what is taken from a list in order to search it. This parameter is important, for example to detect incrementation from an operator to another, as in the example shown in the previous section.

Once an operator and a size have been chosen, we compress the sequence from the left to the right by going as far as possible and appending the result to the previous results until the rest is empty (if the rest is not long enough to be shortened by *size* elements, the program will fail).

Once a list has been compressed, it is rejected if all the operators which have been found are only applied once.

Its complexity is then compared to that of the initial sequence, and the best one is kept (if complexities are equal, we fail).

If we were able to compress until only one operator and its arguments (2 for C and 3 for I) are left, we check that the arguments are coherent and if so, we finally extend it: we increase its *times* and then decompress the final sequence.

I also implemented a way to look at ALL the solutions, but as the adding part has a factorial complexity, this takes a very long time to terminate.

For questions on implementation, see the program documentation as explained in my git repository README[6]. I spent quite a long time writing it as exhaustively as possible, in order to insure a good comprehension of the program.

7 Perspectives

7.1 Increment operators which are applied only once

As seen before, when we find an increment operator which is used only once ($times = 1$), its *where* field doesn't mean anything, as when it is decompressed, only *argument* is extracted, whereas the I, *times* and *where* fields are simply removed from the sequence.

Normally, such a representation should be rejected because trivially, it is less compressed than just writing *argument* as is. It may be useful in this example:

$$1\ 1\ 1\ 2\ 1\ 1\ 2\ 1\ 2\ 3\ 1\ 1\ 2\ 1\ 2\ 3\ 1\ 2\ 3\ 4$$

Let us compress this sequence using our method (again the sequence has been cut for the sake of clarity):

$$1\ 0\ 1\ I$$

$$1\ 0\ 1\ I\ 1\ 0\ 2\ I$$

$$1\ 0\ 1\ I\ 1\ 0\ 2\ I\ 1\ 0\ 3$$

$$I\ 1\ 0\ 1\ I\ 1\ 0\ 2\ I\ 1\ 0\ 3\ I\ 1\ 0\ 4\ I$$

This is of course one of many obtainable solutions, but it is findable by our process so we have the right to consider it. Here we will make an "error" which will cost us the solution:

$$\{\ 1\ 0\ 1\ I\ \}\ 0\ 1\ I$$

$$\{\ 1\ 0\ 1\ I\ \}\ 2\ 2\ I$$

$$\{\ 1\ 0\ 1\ I\ \}\ 2\ 3\ I$$

$$\{\ 1\ 0\ 1\ I\ \}\ 2\ 4\ I$$

Now, we would like to obtain:

$$\{\ \{\ 1\ 0\ 1\ I\ \}\ 2\ 1\ I\ \}\ 5\ 4\ I$$

but our program will fail when comparing 0 with 2.

A solution could be to use a ”joker”, like Factor’s *f* in order to warn, when compared with an other value, that this value has not been defined yet. For now, the program is not able to solve this problem without removing the first isolated 1.

7.2 Masks and indexes

An other way to improve the program would be to solve the ”how to indicate where to apply the increment-operator in a natural, compressed and Kantian way” problem.

This has been a problem since the beginning of the project, and even if many more or less elegant solutions have been found, a lot of work remains to be done on this subject.

7.3 Short term memory

As explained above, the use of short memory is not reliable and quite difficult to implement. Even though a whole implementation of the use of short memory has been done, it has not linked into my final version because of its random behaviour.

An improvement would be to find a way to use it, while keeping this program’s reliability and power.

8 Conclusion

This project has been an excellent way to use, at a project scale, a new and powerful language such as Factor.

It turned out to be difficult to use at first but incredibly efficient for this particular subject. The project itself proved to be harder than expected, because of its ”human” restriction. It has been an opportunity to work and try to contribute solutions to a little-known field.

This was an enriching experience and well worth the time spent working on it.

References

[1] Samuel Tardieu's website: `http://www.rfc1149.net/`

[2] Jean-Louis Dessalles' website: `http://perso.telecom-paristech.fr/~jld/`

[3] A starting point: `http://icc.enst.fr/PLC/Learn.html`, section Complexity

[4] Factor language: `http://factorcode.org/`

[5] Kolmogorov's Complexity on Wikiedia: `http://en.wikipedia.org/wiki/Kolmogorov_complexity`

[6] My git Hub repository: `https://github.com/Rekamux/Free-Project`