Using Universal Processors for the Increase of Evolvability in Digital Ecosystems

Attila Nagy

University of Debrecen
Institute of Mathematics and Informatics
nagyat@dragon.klte.hu

Abstract. Does the evolution work in the digital medium? Can computer programs written in an assembly-like language be evolved? – Despite the significant progress the question remains open. This paper tries to find another missing piece of the puzzle: the evolution of the decoding system. We suggest to apply the idea of the Universal Turing Machine in order to ensure the possibility of open-ended evolution. The first steps of the implementation are also presented.

Almost 10 years have passed away since the birth of `Tierra` [3], which was the first software system for self-replicating evolving computer programs, so called digital organisms. `Tierra` and another richly developed system called `Avida` [1] (first described in [4]) have achieved several valuable results. For example `Avida` introduced some new approaches in the research such as information theory, thermodynamical laws and so on. Unfortunately the results can't be treated as the universal laws of evolvability, for the experiments are not general enough. By the development of `Physis` [2] we tried to provide a general platform for conducting universally valid experiments. `Physis` can incorporate both previously mentioned systems and any other variance while the measurement tools are the same. This feature is necessary for comparative results. The development process shed light on something: this universality is still not enough. We can try numerous fixed architectures in order to provide general results but this way we use only processors which can't evolve. Our main assumption is that the fixed processor structure and instruction set makes the open-ended evolution impossible. First we describe the problem in several aspects then consider some possible solutions. One of them seems to be powerful: the idea of universal processors. We elaborate this concept to some extent.

1 Semantics outside

The existing implementations of digital evolution have the following structure: the genome consists of assembly instructions that describe the behaviour of organisms. The instructions need a decoding system in order to become an actual behaviour (i.e. self-replication, performing computations) as they are pure syntactical symbols. The semantics is not in the genome but in an outer entity –

namely the processor.[7] Without the machines digital organisms are just like viruses. They need a host machine in order to reproduce or to do something.

1.1 Which one is good?

How de we know which decoding system, which processor is suitable for digital evolution? We know to some extent which one is good or actually used in carbon-based life but we can make only rough guesses in the digital world: we create virtual processors just for trying.

Physis tackles this problem as it allows to create and use different processors. By testing several architectures we may get the general properties of virtual machines needed by evolvability. But this possibility – as we will see – is not a significant improvement.

1.2 Fixed versus evolvable decoding systems

The processor is more or less fixed. More or less means that we can use different subsets of the whole instruction set getting different processors but the core architecture of those remains the same.

Although the instruction set can be varied for each experiment it is not changed during the evolutionary scenario. This fact has negative implications. The phenomenon is called *complexity "catastrophe"* [5] When the complexity of organisms increases they tend to be more fragile as the genome-size becomes longer. That's probably because organisms cannot invent new symbolic representations which allow shorter size and the speed of the development gets extremely slow. *The static structure of the decoding systems renders the open-ended evolution impossible.*

2 Possible partial solutions

Here come two possible solutions: one for the preservation of useful sequences and another for providing evolvable processors. Neither are too elegant but they might partially solve the problem and of course they raise many questions regarding the implementational details.

2.1 Frozen instructions

There can be a new attribute for each memory-cell or instruction called *frozen* which means that the location cannot be mutated. This way useful sequences can be locked, preserved. This can be done by the organism itself (via the execution of a special `freeze` instruction) or some other mechanism in the digital ecosystem.

2.2 Instruction allocation

Processors may have a possibility to define new instructions during the run. These instrunctions correspond to subroutines but their codes are not in the organism's genome but in the processor's internal memory area. The organisms take care of defining these new instructions by copying some part of their genome into the processor.

3 The Universal Turing Machine

The very old idea of Universal Turing Machine (UTM)[8] can be applied in digital evolution as an elegant solution for the lack of evolvable decoding systems.

3.1 The definition

The UTM's input tape contains a string with the following structure:

$$m\#c$$

where m is the description of a machine M, c is the code of a program and # is a special marker. The output of the UTM is exactly the same as the output of the M machine started with the c program since it simulates M step-by-step. It's a fundamental model of all possible computations.

3.2 Digital organism: genetic code plus the decoder

Applying the UTM model in digital evolution systems can be done this way: the organsim's genome contains not only its code but its decoding system as well. Of course Turing machines can't be used without modifications since the code would be extremely long and because of the theoretical structure we would loose the connection with the software industry. This way the digital organisms are run on a UTM-like universal processor (UP). The UP first builds the virtual processor needed by the organism and then simply runs the organism's code on it. The organism should replicate fully, it should copy the processor-description as well. The description can be mutated just like the instructions. Something similar happens in natural organisms.

4 The Universal Processor

The idea of universal processors raises several questions.

The universal language of processors How can a machine be efficiently encoded? In theoretical proofs the efficient encoding is not important but here we need an economical method. It may be very hard to construct this descriptive language. How can we construct an ancestor organism with a simple decoding system? We have to make a functional decomposition of self-replicating programs.

Loosing control The idea of universal processors as the underlying physics of the digital world makes the programs even harder to understand. We need very high-level debuggers and semantical analyzers.

These questions can be summarized in one: How can we construct the Universal Processor? In the following sections one possible implementation is presented. It's not yet a description of an existing software but it's probably worth discussing the implementation details. (This is one of the main intentions of this paper.)

4.1 The structure

The structure consists of a continuous memory area, one small (8,16) array of memory cells. The size of the cells is arbitrary (8, 16, 32, 64). 16 seems to be a good choice for alife experiments. The role of the first cell is fixed for all possible processors: it serves as an instruction pointer. The remaining part can be used as registers, stacks or queues.

4.2 The instruction set

The instruction set should fulfill the following requirements:

- each instruction should be as simple as possible:
    - an instruction as elementary building block represents a single action not a compund one
    - an instruction is independent from any addressing mode
- the instruction set should be complete (any more complex operation should be easily definiable)

The instructions are without the type and number of operands. They're defined in the organism's genome. As a first sketch the instruction set is the following:

Data transfer

`in`	reads data from environment
`out`	writes data into the environment
`load`	copy from memory to a structural element
`store`	copy from a structural elements to memory
`move`	general move between structural elements

Control-flow

`jump`	unconditional jump
`ifzero`	skips the next instruction if not zero

Arithmetic and logic

`compare, add, sub, neg, div, mod, shift-l, shift-r, and, or, xor, not ...`

Biological	
`divide`	splits the child and the parent program

Other	
`allocate`	allocates memory
`gfac`	gets the index of the processor currently in facing
`sfac`	sets the index of the processor currently in facing

5 Processor Description Language

5.1 Description of the structure

Each structural primitive has a corresponding symbol and we need a special mark for sectioning:

`R`	register
`S`	stack
`Q`	queue
`B`	blank

For example the `RRSSSBSS` string represents two registers and two stacks with sizes of 3 and 2. (A stack or queue of size 1 acts as a register, so it would be another possible choice to use `SB` instead of `R`.)

5.2 Description of the instruction set

The description of an instruction begins with an `I` symbol after which comes the code of the newly defined instruction. The semantics of the instruction is defined by a microprogram of the *UP*. Shortly:

```
I icode [inst [op1]...]...
```

It can be said that a CISC processor is defined on a RISC architecture. The operands point to structural elements which are indexed by a positive number. (Indices are computed *mod* N where N is the number of structural elements.)

6 Conclusions

The *UP* idea should be implemented and tried out. Of course the "traditional" specific processors can still be used for code-optimization and other simple evolutionary scenarios.

New developments in the hardware industry make the *UP*-concept more feasible. The Crusoe processors are just like universal processors. [6]

References

1. Avida. `http://dllab.caltech.edu`. Digital Life Laboratory.
2. Physis. `http://physis.sourceforge.net`. The homepage of the open-source development.
3. Tierra. `http://www.isd.atr.co.jp/~ray/`. The homepage of Thomas S. Ray.
4. C. Titus Brown Chris Adami. Evolutionary learning in the 2d artificial life system "avida". Kellog Radiation Lab, California Institute of Technology, Pasadena, 1994.
5. Stuart A. Kauffman. *The origins of order - Self-Organization and Selection in Evolution*. Oxford University Press, New York, 1993.
6. Alexander Klaiber. The techology behind the crusoe processors, 2000. `http://www.transmeta.com`.
7. Tim Taylor. Some representational and ecological aspects of evolvability. *EVOLVABILITY WORKSHOP at the Seventh International Conference on the Simulation and Synthesis of Living Systems*, 2000.
8. Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936.