PlasmidPL: A Plasmid-Inspired Language for Genetic Programming

Lidia Yamamoto

Computer Science Department, University of Basel
Bernoullistrasse 16, CH-4056 Basel, Switzerland
lidia.yamamoto@unibas.ch

Abstract. We present PlasmidPL, a plasmid-inspired programming language designed for Genetic Programming (GP), and based on a chemical metaphor. The basic data structures in PlasmidPL are circular virtual molecules or rings which may contain code and data. Rings may react with each other to perform computations on the rings themselves. A virtual chemical reactor stochastically chooses which reactions should occur and when. Code and data may be rewritten in the process, leading to a system that constantly modifies itself. In order to be closer to chemistry, PlasmidPL relies solely on the data and code stored in molecules.
After describing the language, we show some hand-written sample programs that implement initial program generation, mutation and crossover within self-modifying chemical programs. These programs are then used to solve a typical symbolic regression problem, as a feasibility study. Finally, we discuss future directions into specific application scenarios that can benefit from such a chemical model.

1 Introduction

The motivation for this work lies in obtaining software that autonomously reacts to environmental changes by ultimately changing its own code. The applications for such self-evolving software include robotics [1, 2], sensor-actuator networks [3], pervasive, organic and autonomic computing. As in biology, the software itself would be responsible for its own "survival", including reproduction and variation mechanisms which under selective pressure from the real world could result in successive generations of ever improving individuals.

PlasmidPL is a new artificial polymer chemistry [4] inspired by the structure and behaviour of plasmids. In biology, plasmids are small circular DNA segments that can exist and replicate separately from the chromosomal DNA of their host cell which is usually a bacterium. Multiple plasmids and several instances of the same plasmid may co-exist in the same cell.

A PlasmidPL program is a multiset of rings. Rings are circular arrays of atoms that can manipulate other rings, producing new rings as results. A ring data structure wraps around itself in a modulo fashion, such that any position in a ring is a valid position. In this way, information can always be extracted from rings, or written to them without any "array out of bounds" exception.

pre-print of paper to appear in:
Proceedings of the 11th European Conference on Genetic Programming,
EuroGP 2008, Naples, Italy, March 2008, pp. 337-349 (poster presentation)
Springer LNCS 4971, O'Neill et al. (eds.)

As with protected division in GP, the ring structure aims at helping to obtain valid programs using genetic operators. Furthermore, the language is designed with minimal syntactic and semantic constraints, such that any random program can potentially be interpreted to produce some result. In contrast with related approaches to the evolution of programs using chemical metaphors [5, 6], computation with PlasmidPL relies exclusively on the data and code stored in molecules. Any kind of information storage outside molecules (such as external stacks, registers, or memory positions) is explicitly forbidden. The state of the system is therefore fully defined by the set of molecules present in the reactor.

Rings may be regarded either as passive data molecules or as standalone mini-threads of computation. They may react with each other to perform computations on the rings themselves. A virtual chemical reactor chooses which reactions should occur and when, acting as a thread scheduler whose scheduling algorithm emulates a stochastic chemical reaction process. Code and data may be rewritten during the reaction process, threads may fork and join, giving rise to a dynamic system in which code and data are constantly being modified. Indeed, PlasmidPL is heavily based on self-modifying code: due to the absence of explicit variables and external data structures, programs must rewrite themselves in order to get the right values in the right places where they are needed.

In this paper we present the syntax and behaviour of PlasmidPL programs, and show how they can be used to produce code that rewrites itself in order to implement steps from evolution runs such as initial population generation and genetic operators. We then show how these elements can be used in a simple symbolic regression problem: although it looks like a fairly classical GP run, an important difference must be highlighted: the code generation and modification operators act on the same program (or plasmid "soup") where they are located. This is a first step towards evolving self-modifying programs and their genetic operators as done in [7, 8], this time using a chemical metaphor closer to nature.

After a brief introduction to biological plasmids in Sect. 2 and some literature review in Sect. 3, the language is described in Sect. 4. First GP experiments are reported Sect. 5, and further steps, insights and perspectives in Sect. 6.

2 Biological Plasmids

A *plasmid* is a small DNA molecule that can exist and replicate separately from the chromosomal DNA. It is typically circular and double-stranded. Plasmids are most commonly found in bacteria, but have also been found in eukaryotes. Fig. 1 (left) schematically depicts a set of plasmids inside a bacterium.

The structure of a plasmid comprises its *ori (origin of replication)* region, and a set of genes (Fig. 1 (centre)). The ori region is a nucleotide sequence that unites the two extremities of the DNA. During replication, this region is nicked and the DNA is duplicated starting from there. When the replication process is complete the plasmid recircularizes.

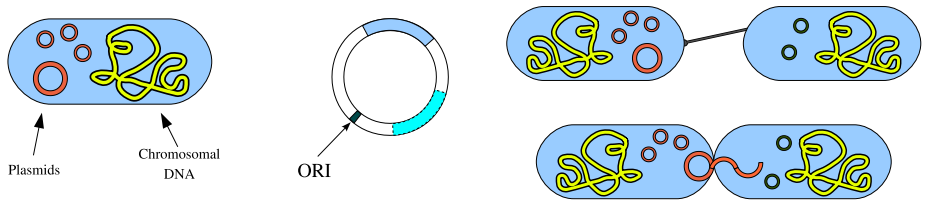

Fig. 1. Left: Bacterium holding chromosomal DNA and plasmids. Centre: A plasmid with its genes (large segments) and its origin of replication (*ori*). Right: A plasmid being transferred from a donor bacterium to a recipient during conjugation.

Plasmids are mobile genetic elements: they often migrate from one bacterium to another via *conjugation*, an asexual mechanism by which genetic material is transferred from a donor cell to a recipient by direct contact (Fig. 1 (right)).

Some plasmid genes may confer selective advantages to the host bacterium, such as antibiotic resistance. Since plasmids may reproduce and migrate to other bacteria, they are important transmission vectors of antibiotic resistance. Plasmids are also extensively used as cloning vectors in genetic engineering. More recently, *Plasmid Computing* has been proposed as a new DNA computing technique that uses plasmids to store information, capable of solving NP-complete problems such as the maximal independent subset problem [9], and the knapsack problem [10].

3 Background and Related Work

In most GP systems the code to be optimized is produced and manipulated by an external program. We would like to explore the possibility of including all the evolutionary steps within the code itself, like in nature, where DNA reproduces with the help of proteins that are manufactured from genes within the DNA itself. Systems with these characteristics have been extensively studied in Artificial Life and Complex Systems research [4, 11, 12]. Many such systems exhibit interesting life-like properties such as self-reproduction [12], self-evolution [6], self-maintenance [4, 11], and so on. They are typically meant to study biological or organizational issues in general, such as the origin of life and the emergence of self-organizing structures. Artificial Chemistries [4] are intimately related to evolution: it is conjectured that evolution itself could have emerged out of catalytic chemical reactions. Modelling programs with an artificial chemistry could perhaps show evolutionary paths that remain unexplored so far.

Artificial Polymer Chemistries [4] are artificial chemistries in which molecules are character sequences that can be concatenated or cleaved during the reactions. PlasmidPL can be seen as a instance of an artificial polymer chemistry, in which molecules contain code that can be transformed via GP. However, our goal is not to study life phenomena but to optimize programs that offer services to users.

The notion of active code strands that operate on passive data strands is present in several earlier artificial chemistries [6, 11, 13, 14]. Similarly, earlier ring-based computation models exist. For instance, in [14] machines operate

on circular tapes that contain a searched pattern, resulting in a system that is able to evolve simple self-replication loops up to complex autocatalytic networks. The potential benefit of PlasmidPL over such systems is to offer a fully blown and intuitive programming language based on a chemical metaphor, in which programs can be produced by humans as well as automatically via GP. Such hybrid approach has been successful in mainstream GP and could bring the chemical metaphor closer to everyday applications.

Our approach share goals with *Ontogenetic Programming* [8] and *Autoconstructive Evolution* [7], both based on variants of the Push language [15]. We took inspiration from Push for some aspects of PlasmidPL, which will become evident in Sect. 4.2. Languages such as MGS [16] are also inspired by a chemical metaphor. However, they have not been designed for GP, and as such, their syntax does not lend itself to easy automatic manipulation.

4 PlasmidPL: Language Description

As in most chemical models, PlasmidPL models programs as a multiset of virtual molecules (a set in which elements may occur more than once). Virtual molecules are rings or plasmids, represented as vectors of indivisible atoms or symbols. All reactions are second-order, i.e. involve two reactants. At each iteration, the reactor selects two molecules at random for reaction: the first one is chosen among the currently active (executable) molecules, and the second among the passive (data) ones.

The symbols in a plasmid are indexed starting from zero at the ori junction point. Any integer index $i \in \mathbb{Z}$ is converted to an index j, $0 \leq j < L$ which always falls within a valid position in the plasmid:

$$\begin{array}{ll} j = i\%L & \text{if } i \geq 0 \\ j = (L - (|i|\%L))\%L & \text{if } i < 0 \end{array}$$

where L is the length of the plasmid in number of atoms and % is the modulo (division remainder) operator. Empty plasmids with $L = 0$ do not make sense and are never present in the multiset, therefore division by zero does not occur. For protected GP, floating point indices are rounded to the nearest integer before the above computation, and other (non-numeric) atoms are treated as zero.

Logically a plasmid wraps around itself in a circular shape. Physically however (in the source code), a plasmid is represented simply as a list of atoms in textual form (a_0 a_1 ... a_{L-1}) equivalent to a LISP list without recursion. The ori position is not represented, and is logically situated between atoms a_{L-1} and a_0. The list format also allows us to refer to plasmid positions informally as *front* positions (near the head of the list) and *rear* positions (near the tail's end).

For computational efficiency in treating long polymers, molecules are identified by *keys* stored in their front symbols. Any atom may in principle play the role of a key. Keys are used to identify reactants, as will be explained next.

Fig. 2 depicts the plasmid reaction metaphor. An active and a passive plasmid react if they have the same key (`x` in the example) at their index positions one

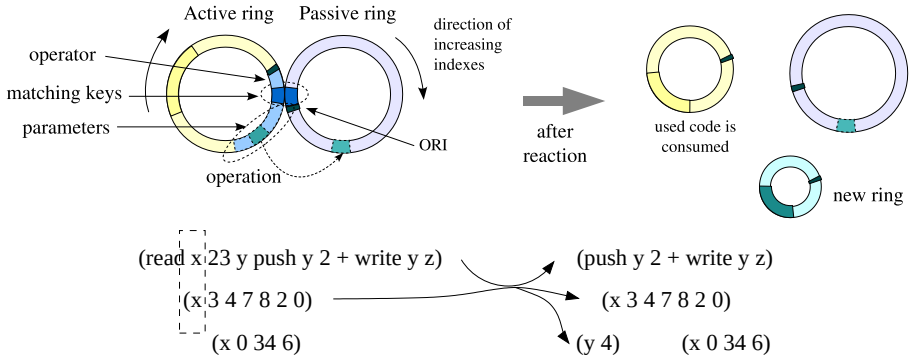

Fig. 2. Reaction between two plasmids. Top: Schematic representation. Bottom: Code example. An active (code) plasmid operates on a passive (data) plasmid.

and zero, respectively. The active plasmid operates on the passive one to obtain reaction products. Both plasmids may be modified in the process, while the executed code of the first (active) plasmid is consumed. The front keyword of the active molecule (i.e. the atom at position zero, `read` in the example) determines the action to be performed on the passive one, according to Reaction Table 1. The action may result in a modification of the passive molecule, of the active molecule itself, and/or the production of another molecule. After the action is performed, the atoms corresponding to the executed code fragment in the active molecule are consumed. In the example of Fig. 2, a `read` instruction produces a third plasmid `(y 4)` containing the read value. `y` is the output key given as a parameter to the `read` instruction, and `4` is the atom at position $23\%L = 2$ of molecule `(x 3 4 ...)` with length $L = 7$. Note from the bottom code example that two passive plasmids with the same key `x` occur: they have equal chance to participate in the reaction, and the first one is chosen at random.

The virtual chemical reactor is driven by a variant of the Gillespie algorithm [17], which simulates the stochastics of a real tank reactor by determining which reactions should occur and when. The choice of reactants is based on the molecule key. This partitions the set of all molecule chains in the reaction into species groups identified by the same key symbol. All the molecules belonging to the same group are treated as the same molecular species as input to the Gillespie algorithm. The search for matching reactants can then be reduced to a simple key lookup in a hash table. Multiple copies of identical rings are represented only once in the reactor by incrementing the *multiplicity counter* (number of copies of an item in the multiset) of the molecule. The resulting algorithm has complexity $\mathcal{O}(s)$ per iteration, where s is the total number of distinct front key symbols.

4.1 PlasmidPL Reactions

The language instructions specify chemical reactions that operate on a molecule whose front atom matches the reaction key. The syntax and semantics of the reaction rules are shown in Table 1.

`read R P S`	read symbol at position `P` from ring `R`; result goes to ring/position `S`
`write R P S`	write symbol `S` at position `P` on ring `R`
`insert R P S`	insert symbol `S` into ring `R` at pos `P`
`delete R P`	delete symbol at position `P` from ring `R`
`join R`	join ring `R` with current ring (append)
`cleave R P`	divide ring `R` between ori and position/character `P`
`copy R`	duplicate ring `R`
`destroy R`	destroy ring `R`
`length R P`	length of ring `R` (written to ring/position `P`)
`exec R`	spawn ring `R` for execution; equivalent to `(copy R delete R 0)`
`push R S`	push symbol `S` onto stack `R`; if `S` is an operator, pop items from `S`, perform the operation and push the result onto `R`
`pop R P`	pops top of stack `R` and writes its value in ring/position `P`
`if C... else...`	if top of stack `C` is true, consume region after `else` atom, else consume region before `else`

Table 1. Semantics of PlasmidPL reactions

The general syntax of a code fragment corresponding to a reaction instruction is (*keyword* k p_1 p_2 ..) where *keyword* is the reserved atom that indicates the type of reaction to be performed (e.g. `read`, `write`), k is the matching key that indicates which molecules may be chosen as passive side for the reaction, and p_i is a set of parameters. Parameters may indicate a position in the passive ring where information should be read/written from/to, the key for a new ring to be produced as output, the specific symbol value to be written to a ring, and so on. As a result of the reaction, rings may join, divide, duplicate, disappear, change from active to passive and vice-versa, become shorter or longer, etc.

For conciseness, we abuse the notation as follows: "ring `R`" means "a ring with atom `R` at position zero", i.e. a ring of the form `(R ...)`. "Ring/position X" refers either to a new ring `(X ...)` (if `X` is a non-numeric atom) or to an offset `X` (if `X` is a numeric value) at the self-molecule (the current active molecule) after the current code fragment (which will be consumed after the reaction). This apparently awkward syntax is intended both to facilitate code rewriting and to ensure that all possible parameters and their types are accepted as valid.

The `read` instruction is a typical example of this double semantics:

`(read a 1 y), (a 10)` → `(y 10)`
`(read a 1 3 write z 1 V), (a 10)` → `(write z 1 10)`

In the first case `(read a 1 y)`, a new molecule `(y 10)` is produced containing the value read from `(a 10)`. In the second case `(read a 1 3)` the same value is written to the self molecule, which can then be used as a parameter for the next instruction in the flow of execution.

An exhaustive description of the instruction set is outside the scope of this paper. Sect. 5 will show some concrete code examples that will hopefully highlight the basic language principles in a practical way.

4.2 Arithmetic and Logic Expressions

We recall that rings are generic data structures that may contain code and data. As such, they can be used to store multiple data structures such as lists, vectors or stacks. Arithmetic and logic expression can be more easily evaluated by looking at a ring as a stack. Three instructions currently have such a semantic: `push`, `pop`, and `if`. This section focuses on the `push` instruction which will be used in the experiments of Sec. 5.3.

When reacting with a matching ring `R`, the `push` instruction uses this ring as a stack: the front portion is the top of the stack and the rear is the bottom. It works as follows: the atom `S` in the self-molecule immediately after the key `R` is inspected: If `S` is an operand (e.g. number) push it to ring `R`. If it is an operator (e.g. `+ - *`), pop needed operands from stack, perform operation and push the result onto `R`. In both cases the corresponding atom is consumed from the active ring. A reaction keyword in the place of `S` (or an empty `S`) acts as a stop condition: the `push` keyword is consumed, together with its key `R`. The stack `R` now contains the result of the computation.

The `push` reaction evaluates a postfix expression like a conventional stack-based language. However the evaluation does not happen at once, but one atom at a time, each time the molecule is chosen for reaction. This preserves the molecule thread model in the sense that other molecules may be chosen for processing in between. The following example shows the reaction path (execution trace) for incrementing a counter `c`:

`(push c 1 +), (c 4)` → `(push c +), (c 1 4)` → `(push c), (c 5)` → `(c 5)`

If there were more instances of `c` molecules in the soup, each instance would have an equal chance to react with `(push c ..)`, regardless of their content, leading to a non-deterministic execution which is very characteristic of chemical models.

The `push` reaction accepts arithmetic, logic and comparison operators which look like those in the C language, plus stack manipulation operators and a number of mathematical functions, including a random number generator. The set of operators can be easily enhanced. The current set is listed in Table 2.

`+ - * / % ^`	arithmetic operators (plus, minus,... mod, power)
`== != > < >= <=`	comparison (equal, not equal, greater than...)
`&& || !`	logic operators (and, or, not)
`swap, dup, del`	stack manipulation: swap the two symbols on top of stack, duplicate or delete symbol on top of stack
`sqrt, log, sin, cos...`	various math functions (square root, logarithm...)
`rnd, int.rnd`	random number generation (float and integer, respectively)

Table 2. `push` reaction operators

For protected GP operations, all the stack operators follow the principle adopted by the Push language [15]: when insufficient arguments are available on top of the stack, or when arguments do not match the required types, or when arguments do not satisfy the pre-conditions for an operation (for example, in case of division by zero) then the operator is interpreted as a *nop* (no operation) and simply discarded without touching the stack.

5 Genetic Programming with PlasmidPL

In this section we show hand-made examples of PlasmidPL programs that implement initial program generation, mutation and crossover in GP. In contrast to most GP systems, these GP operations are performed within the program itself that is being evolved. This is a first step towards evolving code that self-modifies, within which the genetic operators would naturally co-evolve.

5.1 Generating an Initial Program

Here we show a PlasmidPL program that generates a random postfix expression by composing it from a pool of atoms.

```
(genexpr push len dup 0 >
   if len
      push len 1 -
      read atom 1 3
         insert templ 3 A
      exec genexpr
   else
      write templ 0 expr
      destroy len)
```

```
(templ push stack
   write stack 0 result)
(push maxlen dup 1 - int.rnd 1 +
   read maxlen 1 len
   exec genexpr)
(maxlen 6)
(atom 0)  (atom 1) ...
(atom +)  (atom *) ...
(atom dup) ...
```

The left side shows the `genexpr` molecule, which upon activation by an `exec` ring generates an initial expression by combining random atoms taken from the pool of `atom` molecules on the right side. An initialization phase occurs first: the pair `(push maxlen )` and `(maxlen )` react together to produce a molecule `len` containing the target length of the expression to be generated, chosen randomly between one and the maximum length indicated within `maxlen` (6 atoms in this example). After that, `genexpr` is executed: if the length is positive, it decrements it, reads one atom at random from the `atom` pool, and inserts the atom within the template molecule `temp` between the `push stack` and `write` atoms. Then it invokes `genexpr` again, until the target length reaches zero. The template `templ` is then renamed to `expr`, and the temporary `len` molecule is destroyed. `expr` now contains the final expression that will be later evaluated. A sample expression generated in this way is:

```
(expr push stack + 4 5 + / - write stack 0 result)
```

Note that save the fixed prefix and suffix atoms from the template, the expression itself is a random sequence of atoms obeying no syntactic rules. It can nevertheless be evaluated since all the operators are protected.

The same principle could be applied to generate other types of programs, by injecting different `atom` molecules containing the symbols to be composed, e.g. `(atom read)`, `(atom write)`, etc. In this case, however, in order to improve the feasibility of the resulting programs, the modified `genexpr` molecule would have to contain code that inspects the chosen atom and fills in the next atoms with the expected number of parameters. This would be a next step in showing how entirely new code can be produced by rewriting rings.

5.2 Genetic Operators

Code extracts for mutation and crossover are shown below. They operate on the expression `expr` generated as described in Sect. 5.1.

Each `mutate` molecule implements a different type of mutation. For conciseness the choice of the mutation point is omitted, and only the code fragment that actually performs the mutation is shown: the first molecule replaces one symbol at random in the expression, using a `write` to a position `P` which is previously rewritten with the actual index. The value `V` to be written is also previously rewritten by the `read atom 1 3` instruction, which reads one atom at random from those listed in the example of Sect. 5.1. The other `mutate` molecules insert or delete one symbol at random, respectively, in a similar way.

An `(exec mutate)` molecule might react equally likely with each of the `mutate` variants (atom replacement, insertion, and deletion, respectively). The probabilities of each variant may be adjusted by injecting multiple copies of each in different proportions into the reactor.

```
(mutate ...
   read atom 1 3
   write expr P V ...)
(mutate .. insert expr P V ..)
(mutate .. delete expr P ..)
```

```
(crossover
   .. insert cp11 cp12 P
   .. insert cp11 |    P
   cleave cp11 | exec mk1 ..)
(mk1 .. join cp11 join cp22 ch1)
```

The right box above shows fragments of a simple one-point crossover operator. It takes two input expressions, both in the format shown in Sect. 5.1. one of them is the native expression that already resided in the reactor, coming from the initial population generation step. The second one is a copy of another plasmid's expression, injected via a conjugation mechanism. After choosing a crossover point at random in both expressions (which will be written into atom `P`'s position) an identifier for the second segment (`cp12` in the above example) and a breakpoint atom '`|`' are inserted at the crossover point. The rings are then broken up at the breakpoint position and recombined in a crossed way. The result of the crossover is available in molecules `ch1` and `ch2` (not shown), which are later captured for building a new generation as will be shown in Sect. 5.3.

Here is a simplified reaction trace of the crossover mechanism:

```
(cp11 a b c d), (cp21 x y z) →
(cp11 a b | cp12 c d), (cp21 x | cp22 y z) →
(cp11 a b), (cp12 c d), (cp21 x), (cp22 y z) →
(ch1 a b y z), (ch2 x c d)
```

5.3 Postfix Symbolic Regression

The expression generator from Sect. 5.1 together with the genetic operators from Sect. 5.2 can be combined to solve a simple symbolic regression problem. Each individual in the population is a chemical reactor that exchanges molecules with the external environment. The result is a fairly standard stack-based GP

run, except that the code for the generation and modification of individuals is included within the individuals themselves. Besides such code, each individual contains code that: *(i)* obtains an input value corresponding to a fitness case and pushes it onto the stack molecule used to evaluate the automatically generated expression; *(ii)* when the computation is finished, rewrites the stack into an output molecule that is going to be read by an external fitness evaluator. Fitness evaluation and selection are implemented externally, simulating the fact that individuals must survive some environmental pressure. The target function is the well-known quartic polynomial. The fitness function is the sum of the squares of the errors between the obtained and the expected values.

objective:	evolution of a quartic polynomial in postfix form $f(x) = x + x^2 + x^3 + x^4$	crossover prob.:	90%
		mutation prob.:	5%
		selection:	tournament size 4
atom set:	`+ - * / ^ dup swap 0 1 2 3 4 5`	termination criterion:	zero fitness
		max. # generations:	100
fitness cases:	5 values $x \in [-2; 2]$	max. program size:	none
pop. size:	100 individuals	initialization method:	grow

Table 3. Koza tableau of symbolic regression experiments

The Koza tableau for the experiments is shown in Table 3. The population size, maximum number of generations, number of fitness cases and tournament size are intentionally kept small. For simplification no ephemeral constants are used. Moreover, since there are no explicit variables, the program is obliged to take all its input at the beginning when it can be found on the top of the stack.

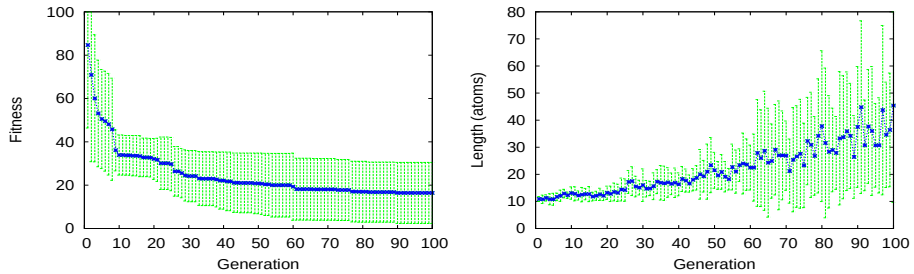

Fig. 3. Evolution of a quartic polynomial using PlasmidPL. Left: average fitness of the best individual of each generation. Right: average length of best of generation.

Fig. 3 shows the evolution results averaged over 10 runs. The average fitness of the best individual decreases with the generations, as expected. Program sizes grow visibly, as well as the variation in sizes. This can be expected as naive mutation and crossover operators were used, with no intron growth control.

One of the runs finds the following 100% correct solution at generation 60:

```
(swap dup dup dup * + swap dup dup dup * + swap 2 ^ * +)
```

It evaluates exactly to: $f(x) = (x + xx) + (x + xx)(x^2) = x + x^2 + x^3 + x^4$ which is the expected expression. It leaves no garbage on the stack and contains only one "intron" (the first swap instruction).

Hand-made solutions include:

```
(dup dup dup 1 + * 1 + * 1 + *)
(dup 4 ^ swap dup 3 ^ swap dup 2 ^ swap + + +)
```

The solution found by GP is parsimonious and at a comparable level of efficiency to hand-made ones. This is remarkable since neither intron growth mechanisms nor stack-correct genetic operators were used.

We have also performed experiments with a quintic polynomial and other functions, with similar results (omitted for conciseness).

6 Discussion

We have described PlasmidPL and shown some examples of how it can be used for a form of GP based on code rewriting. First experiments show that it is feasible to perform fairly "standard" GP with PlasmidPL. This is not the main goal of the language, but only the very first steps. The goal is to have a system that can make use of a chemical metaphor that includes self-modification to control code evolution intrinsically. A research issue in this context is the trade-off between the stochastic nature of program execution, and the potential robustness that could achieved through the redundancy provided by multiple molecules performing similar functions. As pointed out in [5], in an algorithmic chemistry the concentration of instructions is more important than their execution order. A robust online evolution scheme would thus inherently rely on the control of code concentrations.

Experiments with more complex target programs must be performed next. Although helpful, the concept of rings is not sufficient to ensure viable individuals. At the current stage, most instructions expect a fixed number of parameters. This is prone to a *frame shift* in case of a random mutation: a missing or extra parameter may shift the whole execution to a different point, having the effect of a macro-mutation. This can be fixed by allowing a variable number of parameters, combined with protected variation operators that operate on frame borders instead of random positions.

Many aspects of state-of-the-art GP have not been treated here: recursion, modularity, data types, and so on. These should in principle also be possible with a chemical model, however the requirement to operate exclusively on molecules would certainly impose new approaches to these problems.

7 Conclusions and Future Work

We presented PlasmidPL, a new programming language inspired by circular DNA structures called plasmids. The language is still in early stage of development, and is currently only loosely based on real biological plasmids. This paper shows some initial feasibility experiments. PlasmidPL is intended for online tasks related to environment monitoring and control, such as reacting to chemicals present in the environment by diffusing other chemicals in a controlled way.

The chemical metaphor offers a more biologically plausible model of evolution with the potential of autonomous evolution without external support. Indeed, Fontana [11] pointed out that in the physical universe the level of molecules is the only one that "has been observed to spontaneously support complex phenomena

as life". In his chemistry based on λ-calculus, molecules (λ-expressions) represent code and data that operate on each other in a standalone way, thus do not really on a centralized machine architecture. We seek to bring these benefits to practical applications in the GP context. By moving a step closer to chemistry, the building blocks of life, new behaviours might emerge from evolved programs.

8 Acknowledgments

This work has been partially supported by the European Union through IST FET Project BIONETS (http://www.bionets.eu). The author thanks Christian Tschudin, Thomas Meyer, and Daniele Miorandi for their helpful comments.

References

1. Ziegler, J., Banzhaf, W.: Evolving Control Metabolisms for a Robot. Artificial Life **7**(2) (2001) 171–190
2. Taylor, T., Ottery, P., Hallam, J.: An approach to time- and space-differentiated pattern formation in multi-robot systems. In: Proc. TAROS. (2007)
3. Dressler et al., F.: Efficient Operation in Sensor and Actor Networks Inspired by Cellular Signaling Cascades. In: Proc. Autonomics, Rome, Italy (2007)
4. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries – A Review. Artificial Life **7**(3) (2001) 225–275
5. Banzhaf, W., Lasarczyk, C.: Genetic Programming of an Algorithmic Chemistry. In: GPTP II, O'Reilly et al. (Eds.). Volume 8. Kluwer/Springer (2004) 175–190
6. Dittrich, P., Banzhaf, W.: Self-Evolution in a Constructive Binary String System. Artificial Life **4**(2) (1998) 203–220
7. Spector, L., Robinson, A.: Genetic Programming and Autoconstructive Evolution with the Push Programming Language. GPEM Journal **3**(1) (2002) 7–40
8. Spector, L., Stoffel, K.: Ontogenetic programming. In: Proc. Genetic Programming 1st Annual Conf., Stanford University, CA, USA, MIT Press (1996) 394–399
9. Head et al., T.: Computing with DNA by operating on plasmids. BioSystems **57** (2000) 87–93
10. Henkel et al., C.V.: DNA computing of solutions to knapsack problems. BioSystems **88**(1-2) (2007) 156–162
11. Fontana, W., Buss, L.W.: The Arrival of the Fittest: Toward a Theory of Biological Organization. Bulletin of Mathematical Biology **56** (1994) 1–64
12. Sipper, M.: Fifty years of research on self-replication: an overview. Artificial Life **4**(3) (1998) 237–257
13. Laing, R.: Automaton models of reproduction by self-inspection. Journal of Theoretical Biology **66** (1977) 437–456
14. Ikegami, T.: Evolvability of Machines and Tapes. Artificial Life and Robotics **3**(4) (1999) 242–245
15. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: Proc. GECCO 2005, Washington DC, USA (2005) 1689–1696
16. Giavitto, J.L., Michel, O.: MGS: a rule-based programming language for complex objects and collections. Electr. Notes in Theor. Computer Science **59** (2001)
17. Gillespie, D.T.: Exact Stochastic Simulation of Coupled Chemical Reactions. Journal of Physical Chemistry **81**(25) (1977) 2340–2361