University of Amsterdam
Programming Research Group

Molecule-oriented Java Programs for Cyclic Sequences

I. Bethke
P.Walters

Report PRG0201 November 2002

I. Bethke

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7583
e-mail: inge@science.uva.nl

P. Walters

Microsoft
the Netherlands

e-mail: pwalters@microsoft.nl

Programming Research Group Electronic Report Series

Molecule-oriented Java Programs for Cyclic Sequences

Inge Bethke [a,1] Pum Walters [b,2]

[a] *University of Amsterdam, Faculty of Science, Programming Research Group*

[b] *Keyconsult (Keyconsult is a trademark of Babelfish BV, The Netherlands)*

Abstract

We present a theory of finite and repetitive sequences in the conceptual programming style, using molecular programming as a model. We present this theory both in PGLEcm-mppv and in Java.

Key words: program algebra, object, molecule, focus, field, Java.

1 Introduction

In this document we present a theory, in the conceptual programming style, of finite and cyclic sequences. Such sequences arise as a data type in program algebra [4, 5] representation of rational numbers and can serve many other purposes as well. We will take the relevance of sequences with repetition for granted below.

We follow the line of thought in [1] where conceptual programming is put forward as a meaningful objective. Conceptual programs are meant to be read by human readers and contain a conceptual analysis of some topic. At the same time a conceptual program is executable. Its execution is purposefully related to the working of the concept under analysis. The concept can thus be 'interrogated' to exhibit its properties, which are otherwise not formally established.

Our objective is to first present our theory in the frame work of the program algebra PGLE. In this way, PGLE serves two roles: one as a design tool offering

[1] E-mail: inge@science.uva.nl
[2] E-mail: pum@babelfish.nl

a formal foundation of our theory and a second as a tool to understand Java programs. Working then in Java [7], conceptual programming leads to a Java presented theory (JPT, in the terminology of [1]) which is more accessible for a programmer or non-logician than a PGLE presented theory.

The remainder of this document is structured as follows. In Section 2 we discuss molecule-oriented representations of finite and cyclic sequences . Then we offer our theory in the context of program algebra (a PGLEcm-mpp presented theory, or PPT, to coin a phrase) in Section 3 and we offer a JPT in Section 4. Execution of the PPT was used to generate various pictures. Finally, we conclude with some observations in Section 5.

2 Molecule-oriented representations of sequences

In molecule-oriented programming [1] emphasis is on a geometric (graph theoretical) understanding of the structure of objects in a computer memory, as well as of their evolution, during program execution.

A *molecule* is a graph with *atoms* (objects) as its nodes and *instance fields* as its edges. The collection of all atoms, including foci (external named references to atoms) and fields, is called the *fluid*. Although we will not go into details, we mention that molecular dynamics covers the concept of memory management and garbage collection: atoms (and their fields) that are unreachable from any focus are garbage.

Sequences are perhaps the simplest of structuring mechanisms, allowing for arbitrary large molecules. In real life, sequential molecules occur in many places: nylon, the fibers in trees, and DNA.

A sequence molecule is a finite set of atoms, where each atom has one or two fields, one named *entry* and the other (if it is present) named *next*. Each atom, possibly except one, is bound via *next* to another atom in this set, and, traversing only *next* fields, it is possible to start in one of the atoms and visit all other atoms. The *entry* field contains an arbitrary atom—and thus the molecule consisting of all atoms reachable by the atom contained in the field. Occasionally we will distinguish the structural atoms from the content of the sequence molecule. Note that a structural atom may occur as content, so we distinguish not a property of the atom but rather its role in our discussion.

The first picture that may come to mind is Figure 1: In this picture, as in the one below, a • represents an atom and field names are written next to the fields; ⊡, ⊠, . . . denote molecules. If every atom is bound via *next* a cycle must occur, as shown in the Figure 2.

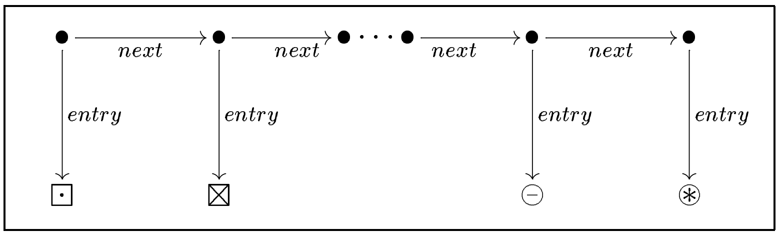

Fig. 1. A finite sequence

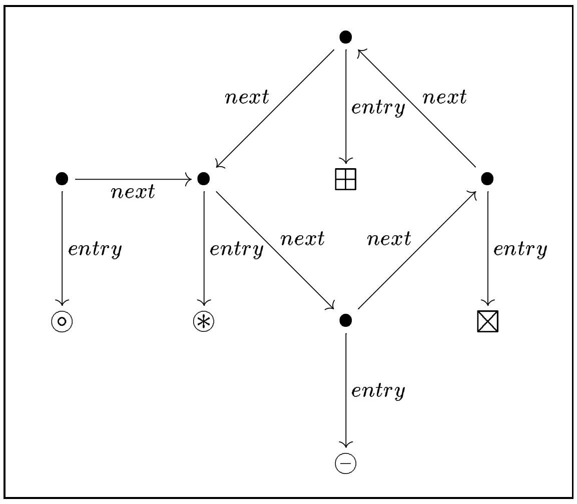

Fig. 2. A lasso-shaped sequence, representing an infinite repetition from the second entry onwards

A sequence molecule defines a possibly infinite sequence of entries. They can be run over by following the *next* field. If a structural atom has no *next* field, the sequence is finite. Otherwise, the sequence becomes repetitive from some point onwards: the molecule contains a cycle (and is called cyclic). We shall refer to molecules in which all atoms are contained in the cycle as circular; we refer to non-circular cyclic molecules as lasso-shaped.

An atom can occur in more than one sequence. Even though our theory is complicated significantly because of this, it is a fact of programming practice that shared sub-sequences occur. Any theory which prohibits this, would be at odds with this reality.

As a matter of fact, any non-empty sequence molecule leads to multiple sequences: each atom can be taken as the first. A sequence is uniquely defined by a molecule and the initial atom. Note, hoewever, that different molecules can represent the same sequence. For example, a cycle can be unfolded any number of times.

In our final example in Figure 3, we will use sequences to represent two decimal expansion of rational numbers[3] with one digit before the decimal

[3] This representation offers true 'infinite precision'. Most programming languages

point (thereby avoiding having to represent that in our example). The content of these sequences is formed by digits. The sequence referenced with focus *1st2207* represents the rational number 22/7 which is a fair approximation of π (4.10^{-4} off). Its decimal expansion is repetitive after one digit with a cycle size[4] of 6. The sequence in focus *1st4407* is the expansion of 44/7 (approximation of $2 \times \pi$). Note that we use value fields—fields carrying next to a name also a type indication and pointing to a label— to denote digits.

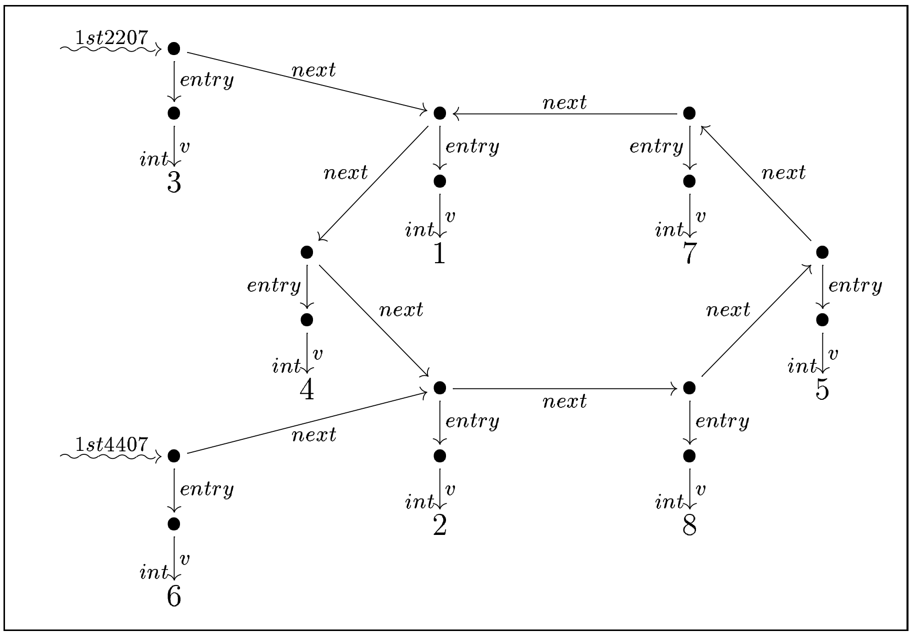

Fig. 3. Two sequences with a shared cycle.

3 Sequences designed in PGLE

Molecular dynamics [2] offers a simple theory, leading to primitives to describe the process of molecule creation and evolution. Basic actions consist of:

(1) creation of an atom;
(2) addition or removal of a field;
(3) effectuation of a field;

offer 'fixed precision' arithmetic (e.g., 32-bits). Some languages (Perl, Python, ...) offer 'arbitrary precision', which runs out when memory runs out. By no means do we suggest this infinite precision representation to be suitable. As mentioned, identical numbers have different representations (even apart from the distinction between, for example, 1.0 and 0.999...). In reality the numerator-denominator representation is probably superior.

[4] A better approximation of π is 355/113, (8.10^{-8} off), and a cycle size of 84 after the first digit.

(4) setting of foci;
(5) tests as to the existence of fields or the equality of atoms.

Basic actions return a boolean value which either depends on the appropriateness of the instruction in the case of mutations and assignments ((1)–(4)) or is the result of a test as in (5).

Evolution of a molecule entails the execution of these basic actions, and any set of molecules can be described by an evolution that brings them about. The sequential execution of these basic actions is formalized by program algebra.

PGA ([4],[5]) is an algebra for sequential programs, focusing on what is traditionally called *control flow*. PGA abstracts from data, assuming all data manipulation to be managed by the set of *basic instructions*. The sole link between data and control flow is embodied in the assumption that every basic instruction returns a boolean value which may or may not be examined by the program.

Base PGA programs are non-empty, possibly infinite sequences of so-called *primitive instructions*:

- perform a basic action—e.g. one of (1) to (5)—and do one of the following three: (i) disregard the boolean value that is returned; or continue with the next instruction if the returned value is (ii) true, or (iii) false, and skip the next instruction and continue with the subsequent instruction otherwise;
- terminate execution;
- continue execution at the n-th subsequent instruction (goto).

There are no backward jumps in base programs.

On top of PGA, the concept of projections is introduced. A projection involves a language and a map which projects programs in that language to base programs under preservation of behavior (behavior roughly being defined as the sequence of actions being performed given the boolean result of those actions). Various languages are defined in [5], among which, most notably, PGLE, which has labels and the jump-to-label instructions. In addition PGLE adheres to the following constraint: the instruction immediately following a conditional is either termination or a goto instruction. Thus, the "then-part" cannot fall through into the "else-part".

In this section we use the language PGLEcm-mpp, which is based on PGLE, with additional control instructions (if-then-else and method call), and in which the basic instructions are precisely the basic actions of molecule oriented programming. A toolset simulating PGLEcm-mpp and other languages can be found on [6]. This toolset can be used to compute projections between program algebra based languages and to generate dynamic representations of molecules.

We have seen that a sequence molecule generally defines more than one sequence; only with the identification of an initial atom is one specific sequence distinguished. In addition to the structural atoms that represent the ordering of elements in the sequence, we introduce a class to represent specific sequences. Atoms in this class have a field *first* with the first structural sequence atom. We refer to atoms in this class as *sequence handles*.

Whether the sequence represented by a handle is finite depends on whether the sequence molecule contains an atom without *next* field. Following the *next* field until we encounter such an atom identifies finite sequences, but not infinite sequences. It would require us also to compare each atom to all previously visited atoms until either an absent *next* field is identified, or an atom is encountered twice (and a cycle is established). Although functionally sufficient this approach is practically unsuitable. Instead, we extend the handle with a *last* field which contains either the final atom (in case of finite sequences), or to the atom immediately before the first repeated atom. Then, a sequence is finite precisely if *last* has no *next*. With this, we can interpret our first generated picture. It represents the same sequences as Figure 3.

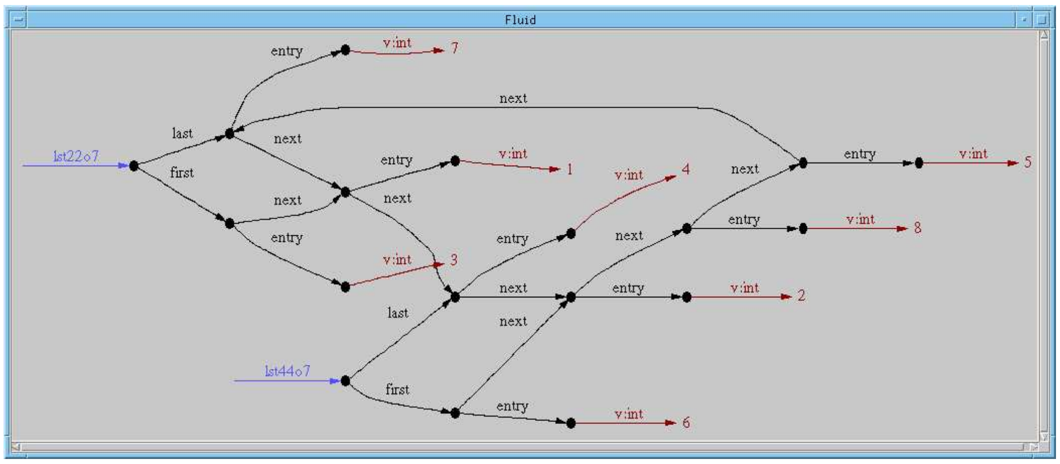

Fig. 4. Figure 3 generated by the PGA toolset.

Part of the program used to generate this picture is shown here:

```
e = new;
e.+v:int = 7;
lst = newSMEl(e);
e = new;
e.+v:int = 5;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
e.+v:int = 8;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
e.+v:int = 2;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
```

```
e.+v:int = 6;
lst44o7 = newSMEl(e);
lst44o7.chain(lst);
e.+v:int = 4;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
lst44o7.last = lst.first;
e = new;
e.+v:int = 1;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
lst = lst.cycle();
e = new;
e.+v:int = 3;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
```

A PGLEcm-mpp program consists of a sequence of basic instructions and control instructions. Basic instructions include the creation of a new atom (`new`), setting of foci (`a = ...`), adding of (value) fields (`a.+b` or `a.+b:int`) and setting them (`a.b = ...`). Field introduction and setting can be combined. In this example control instructions consist of static and non-static method call.

The program fragment above repeatedly creates an entry by calling the method `newSMEL`, creates a list containing only that element, and chains (concatenates) it to the list so-far. At the appropriate point the list for 44/7 is attached, and elsewhere the loop is closed.

Perhaps the most distinctive feature of a sequence is the ability to obtain its first element (`head`) and the remaining elements (`tail`). This is essential in order to process sequences recursively.

Clearly, the head of a sequence is the value bound to the first element. Whereas the tail of a finite sequence is also obtained trivially, cyclic sequence molecules pose a slight challenge. Consider the molecule in Figure 5, in which $X = 3(21)^*$ (where * signifies repetition) is shown, together with its head, tail, and the tail of its tail.

The methods `head` and `tail` are listed below:

```
head() {;
  r = this.empty();
  + r == true {; that = NULL; }{; that = this.first.entry; };
};
```

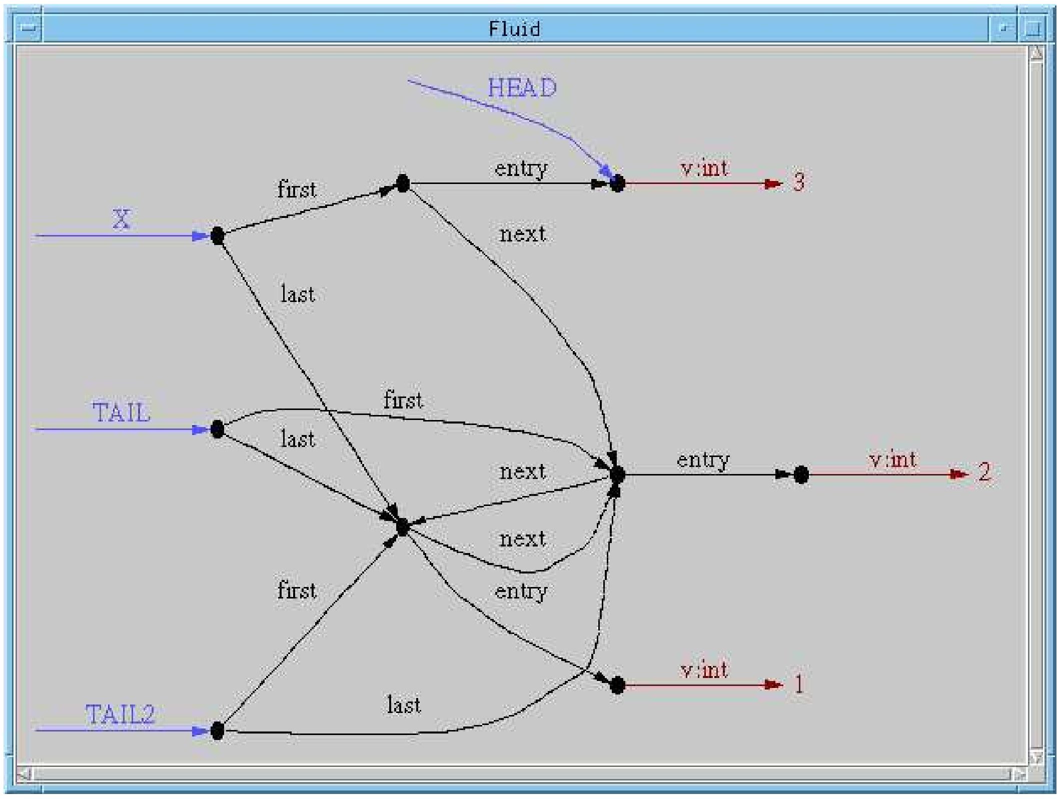

Fig. 5. A generated lasso-shaped sequence with head and tails.

```
tail() {;
  local(result);
  r = this.empty();
  + r == true {;
    result = newSM();
  }{;
    r = this.finite();
    + r == true {;
      result = newSMSMes(this.first.next,this.last);
    }{;
      r = this.circle();
      + r == true {;
        result = newSMSMes(this.first.next,this.first);
      }{;
        result = newSMSMes(this.first.next,this.last);
      };
    };
  };
  r = result;
  result = prev();
  that = r;
};
```

Here, new control instructions are method definitions which—if a result is returned—provide a result in `that`, and the conditional

```
+<exp>{; <when true> }{; <when false> };.
```

PGLEcm-mppv has no local variables apart from `this`, `that`, and parameters. For that reason, we have an explicit stack (a molecule) which temporarily holds the value `result` using the `local` method, and return it using the `prev` method. In our programs we use `r` as a scratch variable.

The method `newSMSMes` generates sequence molecules with `first` and `last` determined by its two arguments (see Appendix A). The properties `empty`, `finite` and `circle` are trivial:

```
empty() {;
  - this/first {; that = true; }{; that = false; };
};

finite() {;
  r = this.empty();
  + r == true {; that = true;
  }{;
    - this.last/next {; that = true; }{; that = false; };
  };
};

circle() {;
  r = this.finite();
  + r == true {; that = false;
  }{;
    + this.last.next == this.first {; that = true; }{; that = false; };
  };
};
```

The real challenge is the definition of the `chain`. Obviously, it is not sufficient to simply bind (with `next`) the last of one to the head of the other argument. If the first argument is cyclic the second argument shouldn't be considered at all, and the first argument should be returned! There is, however, a more subtle concern: what if the arguments share a sub-molecule. Consider the two molecules in Figure 6, the right-hand side being the result of applying the instruction `X.last.+next = Y.first;` in the left fluid.

A proper definition of `chain` must decide to clone (copy structural atoms) under certain conditions. The question under which conditions cloning should take place doesn't have one clear answer. Safest is to clone always, but in

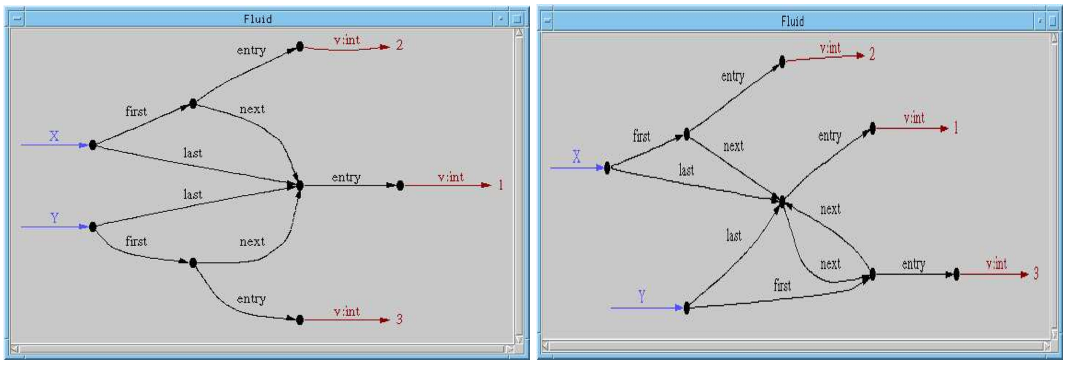

Fig. 6. Before and after: how not to chain.

practice, this is often regarded as being too expensive[5]. Without suggesting our choice to be better than any other, we propose the following: if the first argument is non-cyclic (otherwise it is the chain itself), the shared sub-molecule is cloned, and the atoms in the shorter non-shared segment are reused (i.e. altered). The rationale is that this produces least garbage (unused atoms) and least assignments, and that the handles of the arguments continue to represent the original sequence even though the molecule may have been changed. Note that this approach is invalid if other processes manipulate the structural sequence atoms. This is mainly relevant in the context of parallelism, which is not considered here.

Below is our definition of `chain` and its auxiliary methods. Method `spans` determines whether `this` is longer than `x`, and `diffl` computes a sequence the size of which is the difference in sizes of the arguments to chain (PGLEcm-mpp has no built-in integer arithmetic, but in fact a definition based on integers would be very similar). Method `auxclone` clones the molecule from the point of sharing onwards, and resets the next field otherwise.

```
chain(x) {;
  r = this.empty();
  + r == true {;
    that = x;
  }{;
    r = x.empty();
    + r == true {;
      that = this;
    }{;
      r = this.finite();
      - r == true {;
        that = this;
      }{;
        + this.last == x.last {;
```

[5] Referential transparency suggest copying always, but molecular programming is imperative in nature.

```
          r = this.first.spans(x.first);
          + r == true {;
            r = this.first.diffl(x.first);
            x.first = x.auxclone(r,x.first,this.first);
          }{;
            r = this.first.diffl(x.first);
            this.first = this.auxclone(r,this.first,x.first);
          };
        };
        that = this;
        that.last.+next = x.first;
        that.last = x.last;
      };
    };
  };
};
spans(x) {;
  - x/next {;
    that = true;
  }{;
    + this/next {;
      that = true;
    }{;
      that = this.next.spans(x.next);
    };
  };
};
diffl(x) {;
  - this/next {;
    that = x;
  }{;
    - x/next {;
      that = this;
    }{;
      that = this.next.diffl(x.next);
    };
  };
};
auxclone(d,p,q) {;
  local(pclone);
  + d/next {;
    r = this.auxclone(d.next,p,q.next);
  }{;
    - p == q {;
      r = this.auxclone(d,p.next,q.next);
      p.next = r;
      r = p;
    }{;
      pclone = newSMe(p.entry);
      - p/next {;
        this.last = pclone;
      }{;
        pclone.+next = this.auxclone(d,p.next,q.next);
      };
      r = pclone;
    };
  };
  pclone = prev();
  that = r;
};
```

Method `clone` now can trivially be based on `auxclone`. Care must be taken to temporarily remove cycles. The code is shown in Appendix A, included in the full code for our sequences.

4 Sequence molecules in Java

We shall present all Java programs as JCFs (Java Class Families). The use of JCFs has been advocated in [3] and is explained in detail in [1]. The purpose of the JCF notation is to provide a completely unambiguous account of the way Java sources are stored in a file system, which is relevant because the location of Java sources is part of their meaning. In the present paper only file names will play a role, directories and packages are not taken into consideration.

A JCF is a set of class description files. Each class description file has a name (e.g. `myclass.java`) and a content. The content is an ASCII text containing the source text of one or more Java classes. With JCF-notation it is possible to have several different class descriptions with the same name in one document without causing confusion.

The first JCF contains a class `s` from which all other programs are activated as well as an abbreviation for console output actions. JCFsp = `file:s.java`(

```
class s {
   public static void main(String x[]) { (new c()).m(); }
}
```

) $\cup$ `file:co.java`(

```
public class co {
   public static void p(int x) { System.out.println(x); }
}
```

)

By convention we present extensions of JCFsp to larger JCFs comprising a class `c` with a static method `m()`. Molecule handles are defined by the class `SMj` (sequence molecules in Java). Sequence molecules allow the representation of finite and periodically infinite sequences of arbitrary objects. The repetitive structure inside a sequence results from combining atoms of class `SMe` (sequence molecule-element).

The class definitions for `SMj` and `SMe` are combined in a single source file: JCFseq = `file:SMj.java(`

```
class SMe {                 // sequence molecule element
   Object entry;            // contains the object placed in the element
   SMe next;                // contains the next element in the sequence
   protected SMe(Object e) { entry = e; }
}
class SMj {                 // Sequence molecule objects
   private SMe first;       // first points at the first SMe
   private SMe last;        // last points at the last SMe
   private SMj(SMe f, SMe l) { first = f; last = l; }

   public SMj() { first = last = null; }
   public SMj(Object x) { first = last = new SMe(x);}

   public synchronized SMj cycle() {
      if (!empty() && finite()) { last.next = first; }
      return this;
   }
   private SMe auxclone(SMe d, SMe p, SMe q) {  // prerequisite: finite.
      if (d != null) return auxclone(d.next,p,q.next);
      if (p != q) { p.next = auxclone(d,p.next,q.next); return p; }
      SMe pclone = new SMe(p.entry);
      if (p.next == null) {
         last = pclone;
      } else {
         pclone.next = auxclone(d,p.next,q.next);
      }
      return pclone;
   }
   // using ints (with static int size(SMe)) is slightly clearer in
   // plain Java. However, this works just fine:
   private static boolean spans(SMe p, SMe q) {
      return q == null || !(p == null) && spans(p.next,q.next);
   }
   private static SMe diffl(SMe p, SMe q) {
      return p == null ? q : q == null ? p : diffl(p.next,q.next);
```

```
}
public synchronized SMj chain(SMj that) {
   if (this.empty()) return that;
   if (that.empty() || !this.finite()) return this;
   if (this.last == that.last) {  // shared tail
      // aim: clone shared segment;
      // reuse shortest unshared segment
      // i.e. no garbage, least assignments
      // may change this or that
      if (spans(this.first,that.first)) {
         this.first = this.auxclone(diffl(this.first,that.first),
                                 this.first,that.first);
      } else {
         that.first = that.auxclone(diffl(that.first,this.first),
                                 that.first,this.first);
   }  }
   last.next = that.first; last = that.last;
   return this;
}
public synchronized SMj Clone() {
   if (empty()) return new SMj();
   SMe keep = last.next; last.next = null;
   SMj result = new SMj(first,last);
   result.first = result.auxclone(null,first,first);
   SMe p = result.first, q = first;
   while (q != keep) { p = p.next; q = q.next; }
   result.last.next = p; last.next = q;
   return result;
}
public synchronized Object clone() { return this.Clone(); }
public synchronized Object head() {
   if (empty()) return null;
   return first.entry;
}
public synchronized SMj tail() {
   if (empty()) return new SMj();
   if (finite()) return new SMj(first.next,last);
   if (circle()) return new SMj(first.next,first);
   return new SMj(first.next,last);  //lasso shaped
}
public synchronized boolean empty() {
   return first == null;
}
public synchronized boolean finite() {
   return empty() || last.next == null;
}
public synchronized boolean circle() {
      return !finite() && last.next == first;
   }
   public synchronized boolean lasso() {
      return !finite() && last.next != first;
   }
   public static SMj glue(Object x[ ]) {
      SMj y = new SMj();
      for (int k = 0; k < x.length; k++) {
         y = y.chain(new SMj(x[k]));
      }
      return y;
   }
   public Object take(int x) {     // precondition: x > 0.
      SMe focus = first;
      for (int i = 1; i < x; i++) { focus = focus.next; }
      return focus.entry;
   }
}
```

)

Using `SMj` we define a class `SD10j` for sequences of digits. Lacking an enumeration type the definition of a class exactly matching the ten digits is somewhat cumbersome. Sequences of digits are then an instantiation of sequences taking only digits as entries. The data type of digit sequences is modeled as an algebra without an empty sequence, and with singleton sequences, sequence concatenation and sequence repetition as its operators.

JCFdigseq = JCFseq $\cup$ `file:SD10j.java`(

```
final class D10j {
   static final D10j d0 = new D10j();
   static final D10j d1 = new D10j();
   static final D10j d2 = new D10j();
   static final D10j d3 = new D10j();
   static final D10j d4 = new D10j();
   static final D10j d5 = new D10j();
   static final D10j d6 = new D10j();
   static final D10j d7 = new D10j();
   static final D10j d8 = new D10j();
   static final D10j d9 = new D10j();
   void p() {
      if(this == d0) co.p(0); else
      if(this == d1) co.p(1); else
      if(this == d2) co.p(2); else
      if(this == d3) co.p(3); else
      if(this == d4) co.p(4); else
      if(this == d5) co.p(5); else
      if(this == d6) co.p(6); else
      if(this == d7) co.p(7); else
      if(this == d8) co.p(8); else
      if(this == d9) co.p(9);}
   private D10j() { };
}

class SD10j {   // Finite and periodic sequences of D10j objects.
   private SMj mol;

   SD10j() { } // This dummy constructor allows extending this class
               // with classes having no explicit constructors.
   SD10j(D10j x) {mol = new SMj(x);}
   private SD10j(SMj x) {mol = x;}

   static SD10j concatenate(SD10j x,SD10j y) {
      return  new SD10j((x.mol.Clone()).chain(y.mol.Clone()));
   }
   static SD10j repeat(SD10j x) {
      return new SD10j(x.mol.Clone().cycle());
   }
   static SD10j glue(D10j x[ ]) {
      return new SD10j(SMj.glue(x));
   }
   static D10j take(int x,SD10j y) {      // precondition: x > 0.
      return (D10j) (y.mol.take(x));
   }
}
```

). The test program: JCFdigseqtest = JCFsp ∪ JCFdigseq ∪ `file:c.java`(

```
class c extends SD10j {
  static void m() {
    SD10j x = concatenate(
                new SD10j(D10j.d3),
                repeat(
                  glue(
                     new D10j[ ] {D10j.d1, D10j.d4, D10j.d2,
                                  D10j.d8, D10j.d5, D10j.d7}
                  )
                )
              );
    for (int i = 1; i < 14; i++) { take(i,x).p(); }
  }
}
```

) produces an initial segment of our sequence $1st2207$ shown in Figure 3:

```
3
1
4
2
8
5
7
1
4
2
8
5
7
```

5 Conclusions

We have presented a theory of finite and repetitive sequences. We have done so in a conceptual programming style, using molecular programming as a model. We have presented this theory both in Java and in PGLEcm-mpp.

Java is widely understood, so the JPT is the more accessible. PGLEcm-mpp is specifically geared toward molecular programming, so the PPT is more concise (in concepts, if not in lines).

Developing a theory are similar activities whether undertaken in Java or in PGLEcm-mpp. The ability to generate pictures during development are an exceedingly useful tool in this phase as they are for the presentation of the theory. Our preference for PGLEcm-mpp is partly based on the availability of such a tool.

Java is not specifically aimed at molecular programming, and various aspects are awkward because of that. Nonetheless we feel that Java is suitable.

References

[1] J.A. Bergstra. Molecule-oriented programming in Java. To appear in *Information and Software Technology.*

[2] J.A. Bergstra and I. Bethke. Molecular dynamics. *The Journal of Logic and Algebraic Programming*, 51(2):193–214, 2002.

[3] J.A. Bergstra and M.E. Loots. Empirical semantics for object-oriented programs. Technical report, Lecture Notes, Programming Research Group, University of Amsterdam, 1999.

[4] J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12: 1–17, 2000.

[5] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *The Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.

[6] B. Diertens. The PGA–ProGramAlgebra website. `http://www.science.uva.nl/research/prog/projects/pga`.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Sun Microsystems, Inc., second edition, 2000. `http://java.sun.com/docs/books/jls/`.

A The entire PGLEcm-mpp code

Below is the entire PGLEcm-mpp code used to generate the fluid in Figure 3. The action `fgc` performs a full garbage collection. Various auxiliary variables are set to `null` before termination to avoid the picture being cluttered.

```
vstack = new;

e = new;
e.+v:int = 7;
lst = newSMEl(e);
e = new;
e.+v:int = 5;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
e.+v:int = 8;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
e.+v:int = 2;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
e = new;
e.+v:int = 6;
lst44o7 = newSMEl(e);
lst44o7.chain(lst);
e = new;
e.+v:int = 4;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
```

```
lst44o7.last = lst.first;
e = new;
e.+v:int = 1;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);
lst = lst22o7;
lst = lst.cycle();
e = new;
e.+v:int = 3;
lst22o7 = newSMEl(e);
lst22o7.chain(lst);

lst = null;
d = null;
e = null;
e1 = null;
e2 = null;
p = null;
q = null;
pclone = null;
keep = null;
result = null;
rClone = null;
x = null;
l = null;
this = null;
that = null;
vstack = null;
stackframe = null;
label = null;
r = null;
fgc;
!;

newSMe(e) {;
  that = new;
  that.+entry = e;
};
newSM() {;
  that = new;
};
newSMSMes(e1,e2) {;
  r = new;
  r.+first = e1;
  r.+last = e2;
  that = r;
};
```

```
newSMEl(e) {;
  l = new;
  local(l);
  l.+first = newSMe(e);
  l.+last = l.first;
  r = l;
  l = prev();
  that = r;
};
empty() {;
  - this/first {;
    that = true;
  }{;
    that = false;
  };
};
finite() {;
  r = this.empty();
  + r == true {;
    that = true;
  }{;
    - this.last/next {;
      that = true;
    }{;
      that = false;
    };
  };
};
circle() {;
  r = this.finite();
  + r == true {;
    that = false;
  }{;
    + this.last.next == this.first {;
      that = true;
    }{;
      that = false;
    };
  };
};
head() {;
  r = this.empty();
  + r == true {;
    that = NULL;
  }{;
    that = this.first.entry;
  };
```

```
};
tail() {;
  local(result);
  r = this.empty();
  + r == true {;
    result = newSM();
  }{;
    r = this.finite();
    + r == true {;
      result = newSMSMes(this.first.next,this.last);
    }{;
      r = this.circle();
      + r == true {;
        result = newSMSMes(this.first.next,this.first);
      }{;
        result = newSMSMes(this.first.next,this.last);
      };
    };
  };
  r = result;
  result = prev();
  that = r;
};
chain(x) {;
  r = this.empty();
  + r == true {;
    that = x;
  }{;
    r = x.empty();
    + r == true {;
      that = this;
    }{;
      r = this.finite();
      - r == true {;
        that = this;
      }{;
        + this.last == x.last {;
          r = this.first.spans(x.first);
          + r == true {;
            r = this.first.diffl(x.first);
            x.first = x.auxclone(r,x.first,this.first);
          }{;
            r = this.first.diffl(x.first);
            this.first = this.auxclone(r,this.first,x.first);
          };
        };
        that = this;
        that.last.+next = x.first;
        that.last = x.last;
      };
    };
  };
};
spans(x) {;
  - x/next {;
    that = true;
  }{;
    + this/next {;
      that = true;
    }{;
      that = this.next.spans(x.next);
    };
  };
};
diffl(x) {;
  - this/next {;
    that = x;
  }{;
    - x/next {;
      that = this;
    }{;
      that = this.next.diffl(x.next);
    };
  };
};
auxclone(d,p,q) {;
  local(pclone);
  + d/next {;
    r = this.auxclone(d.next,p,q.next);
  }{;
    - p == q {;
      r = this.auxclone(d,p.next,q.next);
      p.next = r;
      r = p;
    }{;
      pclone = newSMe(p.entry);
      - p/next {;
        this.last = pclone;
      }{;
        pclone.+next = this.auxclone(d,p.next,q.next);
      };
      r = pclone;
    };
  };
  pclone = prev();
  that = r;
};
cycle() {;
  r = this.empty();
  + r == false {;
    r = this.finite();
    + r == true {;
      this.last.+next = this.first;
    };
  };
  that = this;
};
clone() {;
  local(p);
  local(q);
  local(rClone);
  local(keep);
  r = this.empty();
  + r == true {;
    r = newSM();
  }{;
    + this.last/next {;
      keep = this.last.next;
      this.last.-next;
    }{;
      keep = null;
    };
    rClone = newSMSMes(this.first,this.last);
    rClone.first = rClone.auxclone(null,this.first,this.first);
    - keep == null {;
      p = rClone.first;
      q = this.first;
     L0;
      + q == keep {;
        rClone.last.+next = p;
        this.last.+next = q;
      }{;
        p = p.next;
        q = q.next;
        ##L0;
      };
    };
  };
  r = rClone;
  keep = prev();
  rClone = prev();
  q = prev();
  p = prev();
  that = r;
};
local(x) {;
  vstack.+push = new;
  vstack.push.+pop = vstack;
  vstack = vstack.push;
  vstack.+val = x;
};
prev() {;
  + vstack/val {;
    that = vstack.val;
  }{;
    that = null;
  };
  vstack = vstack.pop;
  vstack.-push;
};
```