The Journal of Logic and
Algebraic Programming 51 (2002) 193–214

THE JOURNAL OF
LOGIC AND
ALGEBRAIC
PROGRAMMING

www.elsevier.com/locate/jlap

Molecular dynamics

J.A. Bergstra[a,b,*], I. Bethke[b]

[a] *Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

[b] *Programming Research Group, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

Abstract

Molecular dynamics is a model for the structure and meaning of object based programming systems. In molecular dynamics the memory state of a system is modeled as a fluid consisting of a collection of molecules. Each molecule is a collection of atoms with bindings between them. A computation is modeled as an evolution of the memory fluid. Evolution of the memory fluid takes place by means of chains of actions. Actions transform the structure of molecules in a way similar to chemical reactions. © 2002 Published by Elsevier Science Inc.

Keywords: Object oriented programming; Molecular dynamics; Program algebra

1. Introduction

Molecular dynamics is a simple theory, mainly consisting of notations and informal semantics, bearing on a particular format for states and state changes for computer programs.

In this theory we view a state as a fluid of molecules. A molecule consists of a number of atoms all reachable from one of the atoms—the root—by sequences of directed links. A directed link from one atom to another atom exists if the former has a so-called *field* containing the latter. The link then can be followed from the atom for which it is a field to the one contained in it. By means of actions causing a change of state fields can be added to and withdrawn from atoms, and contents of fields can be modified. In order to make particular behavioural observations, selected atoms can be brought into *focus*.

We will exemplify molecules with fairly primitive diagrams. Fig. 1 depicts our first diagram consisting of two molecules. Here • represents an atom and $x, y, z, \ldots$ denote selected atoms brought into focus. Such foci may share their contents as in the case of u and z. The curly arrow $\overset{x}{\leadsto}$ links the focus x to its content. The straight arrow $\rightarrow$ repre-

[*] Corresponding author.

E-mail addresses: jan.bergstra@phil.uu.nl, janb@science.uva.nl (J.A. Bergstra), inge@science.uva.nl (I. Bethke).

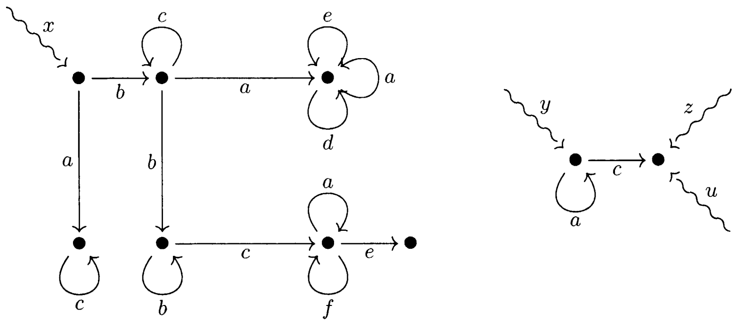

Fig. 1. Fluid consisting of two molecules.

sents a field; $a, b, c, \ldots$ are field names written next to the fields. This fluid contains two molecules, four foci and nine atoms having together 16 fields. The seven irreflexive fields are proper in the sense that they traverse the molecule. The fluid can be presented by means of a *fluid table* as follows:

atoms	1	2	3	4	5	6	7	8	9
fields									
a	6	3	3	4				8	
b	2	7					7		
c		2		5		6	8		
d			3						
e			3					9	
f								8	
foci	x			y	z, u				

In this fluid table $1, 2, \ldots$ enumerate atoms from left to right and from top to bottom. Clearly fluid tables depend on the order of enumeration of the atoms.

There may exist multiple links between atoms. Fields, however, with identical origin, contents and name are identified. The next diagram in Fig. 2 depicts a molecule with

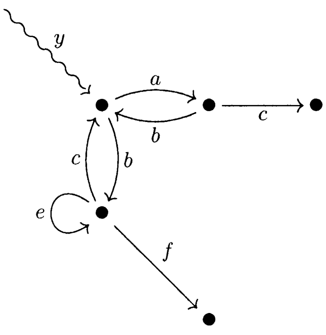

Fig. 2. Molecule with double bindings.

double bindings. Here different atoms are connected in both directions. A fluid table for this molecule is

atoms	1	2	3	4	5
fields					
a	2				
b	4	1			
c		3		1	
e				4	
f				5	
foci	y				

A submolecule visible from an atom is the collection of all atoms reachable by (repeated) field selections from that atom (together with all links between them). A submolecule is maximal if it is not contained in a larger submolecule. A maximal submolecule is also called a molecule. For a focus *x* the molecule visible through *x* is the submolecule visible from the contents of *x*. Thus in Fig. 3, the molecule visible through *x* is the entire molecule, whereas the molecule visible through *y* is the proper submolecule inside the dashed frame. The molecule visible through *u* consists of a single atom.

The fluid making up a state of a computation can be thought of as containing a large collection of proto-atoms standing for memory locations that are still available for accommodating atoms. Creating an atom turns a proto-atom into an atom. The original empty fluid only contains proto-atoms. At the present level of abstraction the number of proto-atoms will not be used. In practice this means that the number is considered unbounded (infinite); in actual computing the initial number of proto-atoms is a measure for the amount of memory space available. The actions in a computation go through a metabolic cycle, starting with the creation of atoms from proto-atoms. Garbage collection completes the

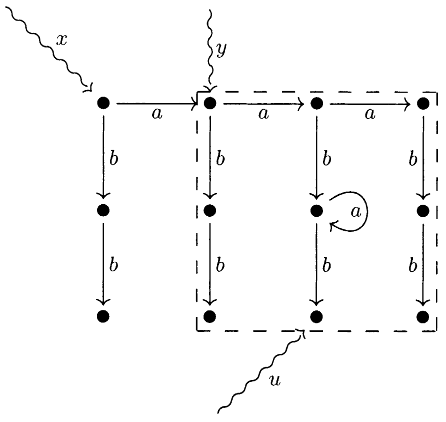

Fig. 3.

metabolism by turning atoms into proto-atoms provided such atoms have garbage status. In Section 6 we will show how to treat proto-atoms graphically.

2. A repertoire of actions and instructions

In molecular dynamics an action is the semantics of an instruction. Instructions are part of programs, actions are part of the evolution of a system.

We consider the following repertoire of eight basic (i.e., indivisible) instructions denoting as many kinds of actions on molecules. Together this is the collection MPP of *molecular programming primitives*. The actions denoted by these instructions are described informally. It is assumed that these descriptions are self-evident. A better way to provide this information might be to use a motion picture. In such a motion picture a slow and continuous motion can introduce a modification of the state while unaffected parts remain firmly in their respective places, thereby conveying a sense of stable identity. For a first step towards a dynamic representation of memory states see [4].

We consider four *mutations*, two *assignments*, and two *tests*. Mutations change a molecule by adding new atoms or bindings and by modifying bindings, assignments modify selected atoms. Both types of instructions return a boolean value depending on the appropriateness of the instruction. Tests return only a boolean value; they do not modify any molecule:

(1) **Mutations**
- (a) Atom creation: Create a new atom and bring it into focus *x*.

  `x = new`

  Atom creation returns the boolean value `true` by default.
- (b) Field introduction: Introduce a *reflexive* field *f* for the atom in focus *x*.

  `x. + f`

  Field introduction returns the boolean value `true` if the atom focussed by *x* does not yet own a field named *f*; otherwise `false` is returned and the instruction is ignored.
- (c) Field withdrawal: Remove the field *f* from the atom in focus *x*.

  `x. − f`

  Field withdrawal returns the boolean value `true` if the atom focussed by *x* owns a field named *f*; otherwise `false` is returned and the instruction is ignored.
- (d) Field mutation: Place the atom focussed by *y* in the field *f* of the atom in focus *x*.

  `x.f = y`

  Field mutation returns the boolean value `true` if the atom focussed by *x* owns a field named *f*; otherwise `false` is returned and the instruction is ignored.

(2) **Assignments**
- (a) Field selection: Bring the atom in member field *f* of the atom in focus *y* into focus *x*.

  `x = y.f`

  Field selection returns the boolean value `true` if the atom focussed by *y* owns a field named *f*; otherwise `false` is returned and the instruction is ignored.

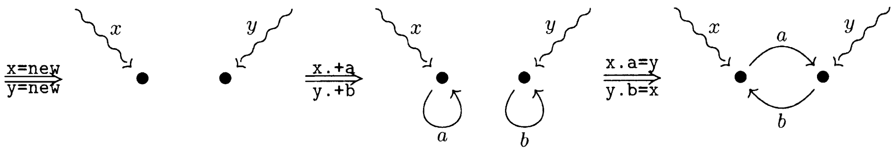

Fig. 4.

(b) Assignment: Place the atom in focus *y* also in focus *x*.

x = y

Assignment returns the boolean value `true` by default.

(3) **Tests**

(a) Atom identity test: This test returns `true` if the foci *x* and *y* share an atom; the test will return `false` otherwise

x == y

(b) Field membership test: This instruction returns `true` in the case that *f* is a field of the atom in focus *x*. Otherwise it returns `false`.

x/f

Note that MPP has two implicit parameters: the collection of focus names and the collection of field names. Both of these collections are assumed unbounded.
As an example consider the action given by the following sequence of basic instructions:

x = new; y = new; x. + a; y. + b; x.a = y; y.b = x

This combination creates a couple of atoms each having one field containing its mate. The diagrams depicted in Fig. 4 render the intermediate actions caused by this sequence on a originally empty fluid.

3. Program notations for programming instructions

In order to express sequences of programming instructions we shall adopt a notational system in the style of program algebra (see [2,3]). There are several such program notations allowing for different degrees of flexibility. The most appropriate notation for molecular programming emerging from our definitions is PGLEc.mpp, i.e., PGLEc using the basic instructions and tests from MPP. Because we will only need PGLEc, we will focus on an introduction of that notation per se.

The syntax of program expressions in PGLEc is generated from nine kinds of constants and one composition mechanism. The constants can be viewed as basic instructions. The composition mechanism is the structuring feature of the programming language: concatenation of *X* and *Y* is written $X; Y$. As a parameter we need a collection of so-called basic instructions: in our case, these are the eight instructions combined in MPP. Thus PGLEc consists of:

(1) *Basic instructions:* All instructions in MPP serve as basic instructions. When executed these instructions may modify a state. After having performed an basic instruction a program must enact its subsequent instruction. If that instruction fails to exist termination occurs.

(2) *Positive test instruction:* For every instruction `a` in MPP there is a positive test `+a`. This test performs `a` and, if `a` returns `true`, then the sequence of remaining instructions is performed; if `false` is returned after `a` was performed, the next instruction is skipped and execution proceeds with the instruction following that skipped instruction. If the respective instructions fail to exist termination occurs.
(3) *Negative test instruction:* For every instruction `a` in MPP there is a negative test `-a`. This test performs `a` and, if `a` returns `false`, then the sequence of remaining instructions is performed; if `true` is returned after `a` was performed, the next instruction is skipped and execution proceeds with the instruction following that skipped instruction. If the respective instructions fail to exist termination occurs.
(4) *Termination instruction:* `!` serves as a notation for an instruction resulting in the immediate termination of the program.
(5) *Label catch instruction:* The instruction Lk, for k a natural number, represents a visible label. As an instruction it is a skip in the sense that it will not have any effect on a state.
(6) *Absolute goto instruction:* For each natural number k the instruction ##Lk represents a jump to (the beginning of) the first (i.e., left-most) label catch instruction in the program which is labeled by the label k. If no such instruction can be found termination of the program execution will occur.
(7) *Conditional instruction:* For every instruction `a` in MPP the instructions `+a{` and `-a{` initiate the text of a conditional construct.
(8) *then/else separator:* The instruction `}{` connects two program sections that are enclosed in braces.
(9) *end brace:* The instruction `}` serves as a closing brace in connection with its complementary opening brace.

The intended meaning of the last three instructions is the following: given any two sequences of instructions $u_1; \ldots; u_n$ and $v_1; \ldots; v_m$, the compound instruction

$$+\mathtt{a}\{; u_1; \ldots; u_n; \}\{; v_1; \ldots; v_m; \}$$

yields the execution of `a` followed by the branch $u_1; \ldots; u_n$ in case `true` was the yield of the instruction `a`, or followed by the second branch $v_1; \ldots; v_m$ in case `false` was returned by the instruction `a`. A compound instruction starting with a negative test selects the first branch in case `false` was returned by the test; otherwise the second branch is executed. After having performed the first or the second branch a program must enact its subsequent instruction. If that instruction fails to exist termination occurs.

Now PGLEc.mpp is a proper subset of all programs obtainable from the nine kinds of instruction using sequential composition. The restriction posed on arbitrary programs justifying their classification in PGLEc.mpp is the following:

each test instruction (*positive or negative*) *must always be immediately followed by a goto instruction or a termination instruction.*

Thus a typical PGLEc.mpp expression is

$$+\mathtt{x} == \mathtt{y}\{; \mathtt{L0}; \mathtt{x} = \mathtt{new}; \}\{; +\mathtt{x/a}; \#\#\mathtt{L0}; \mathtt{x}. + \mathtt{a}; \mathtt{z} = \mathtt{new}; \mathtt{x.a} = \mathtt{z}; \}; !$$

Now let p be a program in PGLEc.mpp. A computation of p can start from the initial fluid S_{init} or can start from some non-empty fluid s. With $p \bullet_{\text{md}} s$ the computation of p on s in the framework of molecular dynamics is denoted. Two things can happen:

(1) The computation properly terminates, resulting in the molecule s'. This molecule is obtained by the succession of instructions prescribed by p. In this case $p \bullet_{\text{md}} s$ denotes s'.

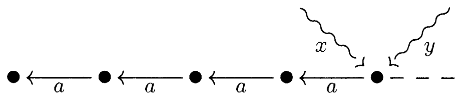

Fig. 5. Limitless evolution.

	$\neg$	$_{\circ}\wedge$	D	T	F	$^{\circ}\vee$	D	T	F
D	D	D	D	D	D	D	D	D	D
T	F	T	D	T	F	T	T	T	T
F	T	F	F	F	F	F	D	T	F

Fig. 6. Truth tables for sequential connectives.

(2) The computation progresses indefinitely. A process of perpetual evolution emerges. This perpetual evolution can lead to a limit which is then denoted by $s' = p \bullet^{\infty}_{\mathrm{md}} s$. The limit exists if each focus remains fixed from some stage onwards, and if all objects get their final collection of fields after finitely many steps in the computation. Fig. 5 depicts a limitless evolution caused by the computation

```
x = new; L0; y = new; y. + a; y.a = x; x = y; ##L0
```

on the initial empty fluid.

All PGLEc.mpp programs are deterministic. It follows that starting from the initial fluid, a terminating or converging program produces a unique molecule or multi-molecule. In which sense is the molecule unique? If in another world the program had been started in the initial state (of that world) as well, a different molecule would have resulted (in a different world). What can be said is that the two multi-molecules (or hypothetical molecules if the whole matter is a mere thought experiment) are isomorphic. A one-to-one connection between their atoms can be made respecting all links. The simplest way to obtain the correspondence is by ordering the atoms of a multi-molecule in their order of creation. This provides an enumeration of the atoms in each fluid generated by a computation starting in the initial state. The isomorphism connects pairwise the atoms that have the same rank number in the respective orders of creation.

4. Assertions and specification

An assertion language is usually needed to formalize properties of the state s of a machine during a computation. In the case of molecular programming assertions must be designed for capturing properties of molecules and fluids.[1]

For the assertion language a 3-valued logic is plausible. We shall adopt the logical framework of [5]. Here the truth values comprise T (true), F (false) and D (undefined). The binary logical operators $_{\circ}\wedge$ and $^{\circ}\vee$ are sequential—which means that evaluation proceeds

[1] Using an algorithmic logic or a process algebra constitutes an alternative for the use of assertions.

from left to right—and are taken from [1]. The matching truth tables are given in Fig. 6. The elementary assertions are as follows:

(1) *Initiation* `init`*:* This asserts that *s* is an empty fluid, the usual starting point for computations.
(2) *Field existence* x/f*:* This assertion takes value T (in fluid *s*) if the atom in focus *x* is currently having a (possibly reflexive) field named *f*. Otherwise the assertion takes value F.
(3) *Identity:* $x.f_1.\cdots.f_n == y.g_1.\cdots.g_m$*:* This elementary assertion returns D if either side is incorrect, i.e., is referencing non-existing fields. It returns T if both sides denote a unique atom in *s*. Finally it takes value F if the sides denote distinct atoms in *s*.

Non-elementary assertions are built from elementary ones by means of negation ($\neg$), (left sequential) conjunction (${}_{\circ}\!\wedge$) and (left sequential) disjunction (${}^{\circ}\!\vee$). An asserted program is a triple of the form

$$[A]\, p\, [B]$$

with *A* and *B* assertions and *p* a program in PGLEc.mpp. $[A]\, p\, [B]$ is *valid* if for all fluids *s*, if *A* evaluates to T in *s*, then $p \bullet_{\mathrm{md}} s$ terminates in s', say, and *B* evaluates to T in s'.

Some examples illustrate this matter. It is assumed that $x, y, z, u, v, \ldots$ are distinct foci. (Of course different foci may share atoms in a given fluid *s*.) The following asserted programs are valid:

(1) $[\mathtt{init}]\ \mathtt{x} = \mathtt{new};\ \mathtt{y} = \mathtt{new}\ [\neg(x == y)]$
(2) $[\neg(x == y)]\ \mathtt{z} = \mathtt{new}\ [\neg(z == x)\, {}_{\circ}\!\wedge \neg(z == y)]$
(3) $[z.f == y.g.h]\ \mathtt{x} = \mathtt{y.g}\ [\neg(z.f == x)\, {}^{\circ}\!\vee\, y.g.h == y.g]$
(4) $[\mathsf{T}]\ \mathtt{z.} + \mathtt{f};\ \mathtt{z.f} = \mathtt{y}\ [z/f\, {}_{\circ}\!\wedge\, z.f == y]$

The following asserted programs are invalid.

(1) $[\mathtt{init}]\ \mathtt{x} = \mathtt{new}\ [\mathsf{F}]$
(2) $[\mathtt{init}]\ !\ [\neg(x.f == y.g)]$
(3) $[\neg(z == y)\, {}_{\circ}\!\wedge \neg(y == x)]\ \mathtt{x.f} = \mathtt{y}\ [\neg(z.f == y)]$
(4) $[\mathsf{T}]\ \mathtt{z.f} = \mathtt{x}\ [\neg(z == x)]$

5. Program families

A *molecular program family* MPF is a collection of programs all of which are supposed to act on the same molecules or fluids. The user of a program family may activate programs in the family in varying order. Often there will be some constraints on the order of usage. For instance there can be a program `initialize` which is to be used only once, preceding all others and so on. An MPF can have named programs or can be made up from a list or can have an enumerated program collection. Below named listed MPFs will be used. The notation employed in the examples is simply taken from set theory.

All of the three following examples concern the representation of natural numbers in molecular dynamics. `Z` will be a constant representing 0 after the first call of the initialization program `setZero`. Except for the initialization programs the other programs represent well-known mathematical operations on numbers. The implicit notational conventions are as follows:

(1) the argument is $\mathtt{x}$ or $\mathtt{x}_1, \mathtt{x}_2, \ldots$ if there is more than one argument,
(2) the result is $\mathtt{y}$ or $\mathtt{y}_1, \mathtt{y}_2, \ldots$ if there is more than one result,

(3) auxiliary foci are `h`, $\mathtt{h}_1$, $\mathtt{h}_2$, ...; they must be initialized within each program,
(4) constants start with a capital.

Natural numbers 1: The first example MPFnn1 contains three programs: `setZero`, `S` and `P`. `setZero` is an initialization program which will create an object representing zero. `P` and `S` compute the predecessor and the successor of a natural number, respectively. In all cases `y` contains the output and `x` contains the input for `P` and `S`. Thus MPFnn1={

```
setZero =def Z = new; y = Z
        P =def −x/p{; y = x; }{; y = x.p; }
        S =def −x/s{; x. + s; y = new; y. + p; y.p = x; x.s = y; }{; y = x.s; }
```

}. A typical molecule—obtained by activating the programs

```
setZero; x = y; S; x = y; S
```

in that order—is depicted in Fig. 7. Here *y* represents the natural number 2.

Observe that the following two asserted programs are valid in each fluid resulting from a succession of the programs in MPFnn1.
(1) $[x == y]$ `P` $[y == x \overset{\circ}{\vee} y == x.p]$
(2) $[x == y]$ `S` $[y == x.s]$
The program family MPFnn1 can be extended. For instance `mod2` computes x mod 2:

```
mod2 =def y = x; L0; +y == Z; !; y = y.p;
             −y == Z{; y = y.p;##L0; }{; y = y.s;}
```

Natural numbers 2: Computing x mod 2 (or equivalently, deciding whether x is even) is 'slow'. A simple redesign of MPFnn1 into MPFnn2 resolves this efficiency problem (at the cost of more memory usage). Each even number object is equipped with a reflexive field *even*, each odd number with a reflexive field *odd*. Thus MPFnn2={

```
setZero =def Z = new; Z. + even; y = Z
        P =def −x/p{; y = x; }{; y = x.p; }
        S =def −x/s{; x. + s; y = new; y. + p; y.p = x; x.s = y;
                         +x/even{; y. + odd; }{; y. + even; }; }{; y = x.s; }
   mod2 =def +x/even{; y = Z; }{; y = Z.s; }
```

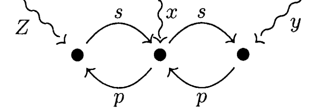

Fig. 7. A typical molecule created by MPFnn1.

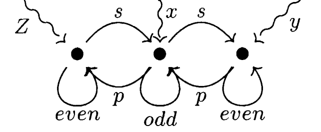

Fig. 8. A typical molecule created by MPFnn2.

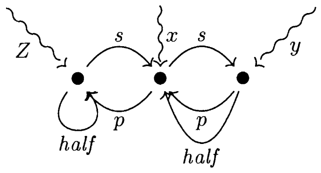

Fig. 9. A typical molecule created by MPFnn3.

}. In Fig. 8 we depict the MPFnn2 version of the molecule depicted in Fig. 7. We extend MPFnn2 by a program which decides the ordering between its arguments. The result `true` is represented by $\mathtt{y} == \mathtt{Z}$ and `false` by $\mathtt{y} == \mathtt{Z.s}$.

$$\mathtt{less} =_{\mathtt{def}} \mathtt{h}_1 = \mathtt{x}_1;\, \mathtt{h}_2 = \mathtt{x}_2;\, \mathtt{L0};\, +\mathtt{h}_2 == \mathtt{Z}\{;\, \mathtt{y} = \mathtt{Z.s};\, \}$$
$$\{;\, +\mathtt{h}_1 == \mathtt{Z}\{;\, \mathtt{y} = \mathtt{Z};\, \}\{;\, \mathtt{h}_1 = \mathtt{h}_1.\mathtt{p};\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{p};\, \#\#\mathtt{L0};\, \};\, \}$$

Natural numbers 3: Again the program `less` in the family MPFnn2 is quite inefficient. The third version of the natural numbers MPFnn3 is a program family admitting fast computation of the ordering relation. Here each even number object is equipped with a field *half* containing the number object representing its half. Fig. 9 depicts the third version of our typical molecule. Thus MPFnn3 = {

$$\mathtt{setZero} =_{\mathtt{def}} \mathtt{Z} = \mathtt{new};\, \mathtt{Z}. + \mathtt{half};\, \mathtt{y} = \mathtt{Z}$$
$$\mathtt{P} =_{\mathtt{def}} -\mathtt{x/p}\{;\, \mathtt{y} = \mathtt{x};\, \}\{;\, \mathtt{y} = \mathtt{x.p};\, \}$$
$$\mathtt{S} =_{\mathtt{def}} -\mathtt{x/s}\{;\, \mathtt{x}. + \mathtt{s};\, \mathtt{y} = \mathtt{new};\, \mathtt{y}. + \mathtt{p};\, \mathtt{y.p} = \mathtt{x};\, \mathtt{x.s} = \mathtt{y};\, +\mathtt{x/half};\, !;$$
$$\mathtt{y}. + \mathtt{half};\, \mathtt{h}_1 = \mathtt{x.p};\, \mathtt{h}_2 = \mathtt{h}_1.\mathtt{half};\, \mathtt{h}_3 = \mathtt{h}_2.\mathtt{s};\, \mathtt{y.half} = \mathtt{h}_3;\, \}$$
$$\{;\, \mathtt{y} = \mathtt{x.s};\, \}$$
$$\mathtt{mod2} =_{\mathtt{def}} +\mathtt{x/half}\{;\, \mathtt{y} = \mathtt{Z};\, \}\{;\, \mathtt{y} = \mathtt{Z.s};\, \}$$

}. A program for computing the order relation is then as follows:

$$\mathtt{less} =_{\mathtt{def}} \mathtt{h}_1 = \mathtt{x}_1;\, \mathtt{h}_2 = \mathtt{x}_2;\, \mathtt{y} = \mathtt{Z};\, \mathtt{L0};\, +\mathtt{h}_1 == \mathtt{h}_2\{;\, \mathtt{y} = \mathtt{y.s};\, \}\{;\, \mathtt{H}_0;\, \}$$

where

$$\mathtt{H}_0 =_{\mathtt{def}} +\mathtt{h}_1 == \mathtt{Z};\, !;\, \mathtt{H}_1$$
$$\mathtt{H}_1 =_{\mathtt{def}} +\mathtt{h}_2 == \mathtt{Z}\{;\, \mathtt{y} = \mathtt{y.s};\, \}\{;\, \mathtt{H}_2;\, \}$$
$$\mathtt{H}_2 =_{\mathtt{def}} +\mathtt{h}_1/\mathtt{half}\{;\, \mathtt{h}_1 = \mathtt{h}_1.\mathtt{half};\, \mathtt{H}_3;\, \}\{;\, \mathtt{h}_1 = \mathtt{h}_1.\mathtt{p};\, \mathtt{H}_4;\, \}$$
$$\mathtt{H}_3 =_{\mathtt{def}} +\mathtt{h}_2/\mathtt{half}\{;\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{half};\, \}\{;\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{p};\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{half};\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{s};\, \};\, \#\#\mathtt{L0}$$

and

$$\mathtt{H}_4 =_{\mathtt{def}} +\mathtt{h}_2/\mathtt{half}\{;\, ;\, \}\{;\, \mathtt{h}_2 = \mathtt{h}_2.\mathtt{p};\, \};\, \#\#\mathtt{L0}$$

Correctness of the program follows from the equivalences

$$2n + 1 < 2m + 1 \Leftrightarrow 2n < 2m$$
$$2n < 2m \Leftrightarrow n < m$$
$$2n + 1 < 2m \Leftrightarrow 2n < 2m$$
$$2n < 2m + 1 \Leftrightarrow n < m + 1$$

6. Garbage: detection, collection and removal

Assuming that a program starts its operation in S_{init}, i.e., no objects exist yet, the concept of garbage is easy to define. Let *m* be a multi-molecule describing an intermediate state of the computation. Atoms of *m* are now classified as garbage or non-garbage according to their reachability by any focus through successive field selection. Non-garbage atoms are inductively defined as the smallest collection of atoms of *m* that satisfies the following two criteria:

(1) the objects in any focus are non-garbage,
(2) if *a* is a non-garbage atom with some field *f* which contains the atom *b*, then *b* is a non-garbage atom as well.

Garbage atoms are the atoms that have not been classified as non-garbage.
Under the assumption that storage of atoms has some cost, the mere existence of garbage atoms is potentially problematic for a computation. Of course in 'reality' only a bounded number of atoms can be stored. Then the problem may surface because the creation of a new object fails by lack of freely available memory space. As it stands PGLEc.mpp has no facility to express any commands regarding garbage removal.

We assume the possibility to store an atom in combination with a so-called *reference count*. The reference count indicates how many references to an atom exist. The reference count is updated with each assignment, mutation, focus or atom removal. Moreover, we assume the existence of a unique atom *null*—i.e. a constant focus *null*—which collects all existing foci which do not explicitly focus atoms different from *null*. Three natural instructions come to mind as an extension of MPP:

(1) Removing a focus *x*: This removal is permitted only if the atom in focus *x* has reference count 1. Its removal is obtained by turning the atom into a proto-atom and putting the atom *null* in focus *x* instead of its original atom. Each atom directly reachable from the degraded atom has its reference count decreased by 1. Notation:

```
rma x
```

(`rma` for `re`move `a`tom.)

(2) Restricted garbage collection: This garbage collection invokes the cumulative removal of all atoms that have reference count 0. After turning such an atom into a proto-atom (see also Section 1) each atom directly reachable from it (by field selection) has its reference count decreased by 1. Atoms thereafter having reference count 0 will be removed by turning them into proto-atoms and so on. Notation:

```
rgc
```

(`r`estricted `g`arbage `c`ollection). Not all garbage will be removed by means of restricted garbage collection: cycles in the reference structure will be left intact.

(3) Full garbage collection: This is an extensive process resulting in the removal of all garbage atoms. Notation:

```
fgc
```

`F`ull `g`arbage `c`ollection will remove cyclic garbage as well as garbage which can be identified on the basis of reference counting.

The different phases of a typical garbage collection process are displayed in Fig. 10. Here field names are omitted and proto-atoms are rendered as circles. It should be noticed that as a program notation PGLEc.mpp+`rma` is unproblematic whereas PGLEc.mpp+`rgc` and PGLEc.mpp+`fgc` are—although being safe—very high-level and will take an amount of time which is hard to predict.

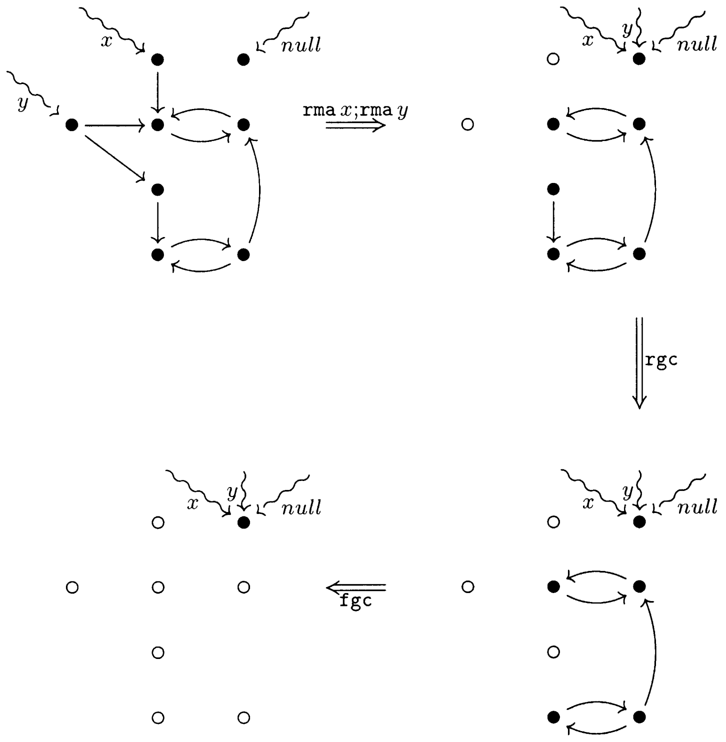

Fig. 10. Garbage collection process in different phases.

7. Values

The exposition of molecular dynamics thus far fails to take notice of the concept of a value. Values are central to most program notations. Nevertheless it is far from easy to explain why values exist and how they differ from objects. Some preliminary remarks can set the stage for a contemplation of values.

(1) Most program notations allow a fixed and finite number of basic types (such as `boolean` and `int`) each containing a fixed and finite number of elements. These elements are values.
(2) Usually for each value there is at least one so-called literal. A literal is an expression denoting the value in a notation most likely to be known ‘outside’ the program notation as well (such as `true`, `false`, `0`, `1`, ...).
(3) From the viewpoint of object-orientation, values are objects for which it cannot pay off to arrange storage indirectly via a reference. The reference is at least as expensive (‘big’) as the object itself.
(4) It is consistent with molecular dynamics to view a value as a terminal object (an object without outgoing arrows) labeled with a literal. Two values p and q are identical if they are terminal objects carrying the same label. Non-value objects do not carry a label but a •.

Fig. 11 gives an intuitive idea of how values can be depicted in a diagram. We will split fields into two categories: those referring to objects remain unchanged; value fields, however, carry a type indication and point to a label. In the present case, there are the

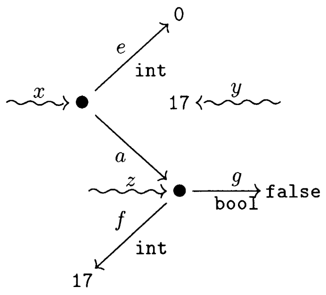

Fig. 11. A molecule with value fields.

three value fields *e*, *g*, and *f*. *e* points to the integer label `0`, *g* points to the boolean label `false`, and *f* points to the integer label `17`. Moreover, values can be put into focus. In our case, the value 17 is focussed by *y*. The collection of instructions MPPV extends MPP with instructions for dealing with values.

(1) *Mutations*. Value mutations change the molecule by adding new values or by modifying them. Field withdrawal works for value fields as well and is not repeated.
   (a) Value field introduction: Introduce a new value field *f* of type *t* for an atom in focus *x*. The value will be initialized by a type dependent label (e.g., `0` in the case of integers, `false` in the case of booleans, etc.)

   `x. + f : t`

   Value field introduction returns the boolean value `true` if the atom focussed by *x* does not yet own a field *f* of type *t*; otherwise `false` is returned and the instruction is ignored.
   (b) Value field mutation: For an atom in focus *x*, replace the label of the value contained in field *f* by the label of the value in focus *y*.

   `x.f = y`

   Here *replace* means copy and *not* share the label. Value field mutations return the boolean value `true` if the atom focussed by *x* owns a field *f* of the appropriate type; otherwise `false` is returned and the instruction is ignored.
   (c) Constant value field mutation: For an atom in focus *x*, replace the label of the value contained in field *f* by `u`.

   `x.f = u`

   Constant value field mutations return the boolean value `true` if the atom focussed by *x* owns a field *f* of the appropriate type; otherwise `false` is returned and the instruction is ignored.
(2) *Assignments*.
   (a) Value field selection: Bring a copy of the value in field *f* of the atom in focus *y* into focus *x*.

   `x = y.f`

   Value field selection returns the boolean value `true` if the atom focussed by *y* owns a field *f*; otherwise `false` is returned and the instruction is ignored.

(b) Value assignment: Put a copy of the value in focus *y* in focus *x*.

`x = y`

Value assignment returns the boolean value `true` by default.

(c) Constant value assignment: Place a value with label `u` in focus *x*.

`x = u`

Constant value assignment returns the boolean value `true` by default.

(3) *Tests*. Just as the tests of MPP these tests do not change the state. The field membership test works for value fields as well and is not repeated.

(a) Value identity test: Return `true` if the values in the foci *x* and *y* carry the same labels and `false` otherwise.

`x == y`

(b) Constant value identity test: Return `true` if the value in focus *x* carries the label `u` and `false` otherwise.

`x == u`

Given the above-listed instructions, one possible program resulting in the multi-molecule depicted in Fig. 11 is the following:

```
              x = new;   z = new;   y = 17;
x. + e : int;   x. + a;   x.a = z;   z. + g : bool;   z. + f : int;   z.f = y
```

8. Projection semantics for method calls

In this section we will discuss the syntax and semantics of a couple of extensions of PGLEc.mpp, beginning with the return and returning goto instruction.

8.1. Recursion

The introduction of recursion on top of PGLEc.mpp can be achieved by introducing two new instructions: the *returning goto* instruction and the *return* instruction. The syntax of these instructions is as follows:

- `R##L`*k*, with *k* a natural number and
- `R`.

The returning goto instruction `R##L`*k* performs a goto together with the command to jump to the instruction immediately following itself whenever a return instruction is reached in the computation following `L`*k*. The return instruction `R` jumps to the instruction just after the last returning goto instruction to which a return has not yet taken place.

As an example consider the following recursive definition of addition on natural numbers. The program assumes the conventions stated in Section 5; `S` is the successor program

```
add=def −x2/p{; y = x1; }
        {; L0; −x2/p; R; x2 = x2.p; x = x1; S; x1 = y; R##L0; y = x1; }
```

A better description of the semantics of these instructions is given in terms of a stack. Thus let us assume that there is a stack of control points (e.g. instruction numbers).

- A returning goto instruction first puts the control point just following it on the stack and then performs the goto instruction.
- A return instruction performs a look-up on the stack and finds the return point on top of the stack. It then issues a goto towards this control point. After that goto has been performed the stack is popped.

We will translate the two return instructions into PGLEc.mpp using this intuition as a guide.

Our translation presupposes the existence of a fieldless atom focussed by *stackframe*. In other words, we assume that every program p is preceded by the instruction `stackframe` $=$ `new`. Moreover, since the translation introduces labels some precaution has to be taken. In order to avoid clashes with already existing labels, we assume that in a given program $u_1; u_2; \cdots; u_n$ ordinary goto and corresponding label catch instructions are labelled by powers of 2; return and returning goto instructions will then introduce fresh labels carrying powers of 3. We now translate the program $u_1; u_2; \cdots; u_n$ into $\psi_1(u_1); \psi_2(u_2); \cdots; \psi_n(u_n)$, where $\psi_i(u_i) = u_i$ if u_i is a PGLEc.mpp instruction, and

- ψ_i(R##Lk) =

```
aux = new;
aux. + back;
aux.back = stackframe;
stackframe. + next;
stackframe.next = aux;
aux. + label : int;
aux.label = 3^k;
stackframe = aux;
##Lk;
L3^k
```

- ψ_i(R) =

```
−stackframe/label; !;
label = stackframe.label;
stackframe = stackframe.back;
stackframe. − next;
##L[label]
```

The focus depending goto ##L[label] in the last instruction of ψ_i(R) abbreviates

$$+\texttt{label} == 3^0; \#\#\texttt{L}3^0; +\texttt{label} == 3^1; \#\#\texttt{L}3^1; \ldots; +\texttt{label} == 3^m; \#\#\texttt{L}3^m$$

where 3^m is the maximal return label on the stack.

A typical stack generated by a couple of returning goto instructions is depicted on the left of Fig. 12. Here $\mathtt{l}_1, \ldots, \mathtt{l}_{\mathtt{j}+2}$ are the return labels. On the right the stack is shown after two return instructions without any intermediate garbage collection.

8.2. Void unparameterized static method calls

The definability of both the returning goto and the return instruction serve as a basis for a projection semantics for method calls. We shall introduce the syntax and semantics for method calls in stages starting with *void unparameterized static* methods which simply perform a computation without returning a value. Throughout this and the following sub-

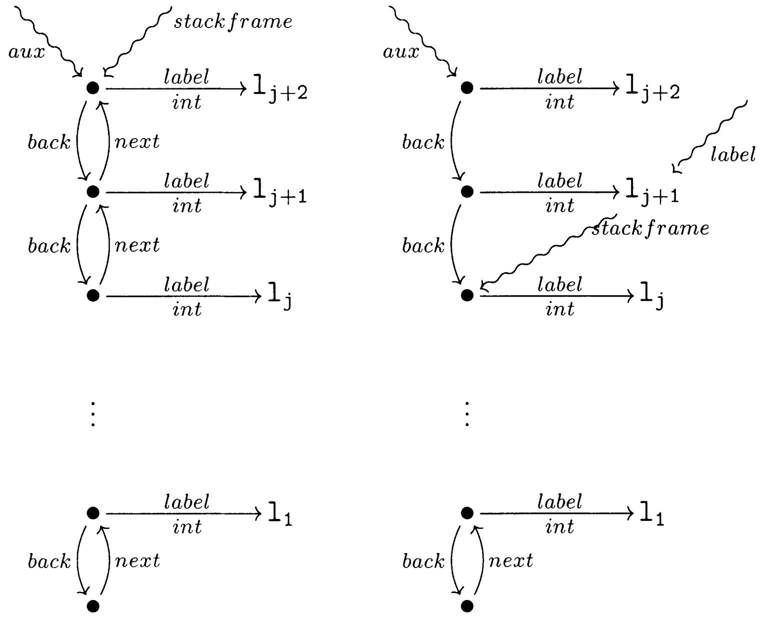

Fig. 12.

sections we work—in object-oriented terminology—in the framework of a single anonymous class, i.e., a local class without a name (see e.g. [6]).

The syntax for declaration and call of void unparameterized static methods are the following:

- $\mathtt{m}k()\{;u_{i+1};\ldots;u_{i+m};\}$ with k a natural number, and
- $\mathtt{m}k()$.

Here $u_{i+1};\ldots;u_{i+m}$ make up the body of the method. In order to avoid clashes of labels during translation, fresh labels carry powers of 5. Given the returning goto and the return instruction, a projection semantics can be given simply by

- $\psi_i(\mathtt{m}k()\{;u_{i+1};\ldots;u_{i+m};\}) = \mathtt{L5}^k;\ \psi_{i+1}(u_{i+1});\ldots;\psi_{i+m}(u_{i+m});\ \mathtt{R}$
- $\psi_i(\mathtt{m}k()) = \mathtt{R\#\#L5}^k$

8.3. Non-void unparameterized static method calls

On a *value-returning* method we impose the restriction of focussing the value to be returned by the focus `that`; thus the body of such a method must contain at least one assignment of the form $\mathtt{that} = \mathtt{y}$. Syntax and semantics are now easily given by

- $\mathtt{m}k()\{;u_{i+1};\ldots;\mathtt{that}=\mathtt{y};\ldots;u_{i+m};\}$
- $\mathtt{x} = \mathtt{m}k()$
- $\psi_i(\mathtt{x} = \mathtt{m}k()) = \mathtt{m}k();\ \mathtt{x} = \mathtt{that}$

The semantics for the declaration of a method is similar to the void case.

8.4. Non-void static method calls with parameters

In order to incorporate parameters, we assume the existence of fixed additional foci $\mathtt{arg1},\ldots,\mathtt{argn}$ which may occur in the body of a parameterized method and which during the call focus its parameters. The temporary storage of the objects originally focussed by $\mathtt{arg1}, \ldots, \mathtt{argn}$ is achieved by the introduction of additional *stackframe* fields $arg1, \ldots, argn$. Thus including parameters requires the following adaptations of syntax and semantics:

- $\mathtt{m}k(\mathtt{arg1}, \ldots, \mathtt{argn})\{;\, u_{i+1}; \ldots; u_{i+m};\}$
- $\mathtt{x} = \mathtt{m}k(\mathtt{v1}, \ldots, \mathtt{vn})$
- $\psi_i(\mathtt{x} = \mathtt{m}k(\mathtt{v1}, \ldots, \mathtt{vn})) =$

  $\mathtt{stackframe.+arg1}; \ldots; \mathtt{stackframe.+argn};$
  $\mathtt{stackframe.arg1} = \mathtt{arg1}; \ldots; \mathtt{stackframe.argn} = \mathtt{argn};$
  $\mathtt{arg1} = \mathtt{v1}; \ldots; \mathtt{argn} = \mathtt{vn};$
  $\mathtt{x} = \mathtt{m}k();$
  $\mathtt{arg1} = \mathtt{stackframe.arg1}; \ldots; \mathtt{argn} = \mathtt{stackframe.argn};$
  $\mathtt{stackframe.-arg1}; \ldots; \mathtt{stackframe.-argn}$

Methods with arbitrary parameters $\mathtt{p1}, \ldots, \mathtt{pn}$ can now easily dealt with using α-conversion. That is, the syntax and semantics of arbitrarily parameterized method declarations can be given by

- $\mathtt{m}k(\mathtt{p1}, \ldots, \mathtt{pn})\{;\, u_{i+1}; \ldots; u_{i+m};\}$
- $\psi_i(\mathtt{m}k(\mathtt{p1}, \ldots, \mathtt{pn})\{;\, u_{i+1}; \ldots; u_{i+m};\}) =$

  $\mathtt{m}k(\mathtt{arg1}, \ldots, \mathtt{argn})\{;\, \psi_{i+1}(u_{i+1}[\vec{\mathtt{p}} := \vec{\mathtt{arg}}]), \ldots, \psi_{i+m}(u_{i+m}[\vec{\mathtt{p}} := \vec{\mathtt{arg}}]);\}$

  where $u_j[\vec{\mathtt{p}} := \vec{\mathtt{arg}}]$ denotes the instruction obtained by substituting $\mathtt{p1}, \ldots, \mathtt{pn}$ by $\mathtt{arg1}, \ldots, \mathtt{argn}$.

The semantics for the declaration of a method is similar to the unparameterized case.

8.5. Non-void instance method calls with parameters

The last step consists of including *calling objects*. A method called for a specific object provides the implementation of the dynamic behaviour of that object and may change its state. In order to model this process we equip the stack in Fig. 12 with an additional field named *this* which is temporarily referencing the current object. The object for which the method is called will then be put into the focus `this` and the method is called. After the call the current object is returned into focus `this`. Thus syntax and semantics for the call of this kind of methods are given by

- $\mathtt{y} = \mathtt{x.m}k(\mathtt{v1}, \ldots, \mathtt{vn})$
- $\psi_i(\mathtt{y} = \mathtt{x.m}k(\mathtt{v1}, \ldots, \mathtt{vn})) =$

  $\mathtt{stackframe.+this};$
  $\mathtt{stackframe.this} = \mathtt{this};$
  $\mathtt{this} = \mathtt{x};$
  $\mathtt{y} = \mathtt{m}k(\mathtt{v1}, \ldots, \mathtt{vn});$
  $\mathtt{this} = \mathtt{stackframe.this};$
  $\mathtt{stackframe.-this}$

Syntax and semantics for the declaration of such methods remain unchanged.

9. Static fields in Java

The definition and initialisation of static fields in the programming language Java is not a trivial matter. Below three Java programs are displayed that produce remarkable results which can hardly be guessed by someone with marginal knowledge of Java only. We will use program algebra and molecular programming to provide ad hoc projections for these programs. As it turns out, projection semantics may be helpful for understanding the behaviour of individual programs.

9.1. Preliminaries

In what follows we assume that running a program from the series of examples below results from the fixed command `java s` which will call `main()` of class `s`. The execution of `main()` consists of a call of a method `m()` of class `c`. All Java programs, when translated into a program algebra based notation, will then be projections of a program of the form

```
make − c(); c.m(); !; X
```

with `X` describing class `c` and other classes when present. As an example, we consider the program

```
class c {
    static void m() {
    co.p(true);
  }
}
public class co {
    public static void p(boolean b) {
    System.out.println(b);
    }
}
```

which—when executed—prints a line containing `true` on the console. In order to project this program a further ingredient is needed: instructions with externally observable (side) effects. Here two such instructions are needed:

- `co-p-true`: print `true` on the console followed by a new line,
- `co-p-false`: print `false` on the console followed by a new line.

Both instructions return the boolean value `true`. A projection of our program can now be given by

```
make-c();c.m();!;
make-c(){;+c == null{;c = new;};};
m(){;f = true;make-co();co.p(f);f = null;};
make-co(){;+co == null{;co = new;};};
p(f){;+f == true{;co-p-true;}{;co-p-false;};}
```

Here each class is represented by an object. Moreover, the class will be named by a focus carrying the name of the class throughout the program. Initially that focus, like all other foci contains the same object as the focus `null`. There will never be made assignments to `null`, it is a constant focus (in the programs that will serve as projections below). A method `make-c()` will be responsible for constructing the object for class `c`, and for generating fields for each of its static fields. Thus, whenever a non-inherited field or a non-inherited method or a constructor of a class `c` is used the method `make-c()` is called just before the instruction that makes the use. `make-c()` checks itself that only a single object representing `c` will be introduced. After that has been done its effect is neutral—the state/molecule is not changed.

The three examples discussed below will all involve the print method in class `co`. For reasons of brevity, we will omit this class in the sequel.

9.2. Circular dependence of static booleans

As a first example, we consider the program obtained from

```
class c {

    static void m(){

    co.p(c1.b1);

    co.p(c2.b2);

  }
}
class c1 {
static boolean b1 = !c2.b2;
}
class c2 {
static boolean b2 = !c1.b1;
}
```

This program produces

```
false
true
```

To obtain a projection the method `make-c()` needs to be more involved in order to take static fields into account. We thus let `make-c()` call two further methods:

- `make-statics-c()` which is responsible for producing all static fields of the class—modeled as fields of the class object—and providing these with a default initialisation (`false` in the case of booleans, 0 in the case of integers), and
- `init-statics-c()`, responsible for computing all non-default initialisations of static fields.

The whole point of this projection is that it brings about the following mechnism of Java. When `init-statics-c()` is executed, that execution may involve a call to `make-c'()` for some class `c'`. When executing this second call, all static fields of `c` can be used, and those fields for which the non-default initialisation has not been completed still have their default initial value.

The detailed analysis along the lines just mentioned yields the following projection:

```
make-c();c.m();!;
make-c(){;+c == null{;c = new;

                    make-statics-c();init-statics-c();};};

make-statics-c(){;};
init-statics-c(){;};
```

```
m(){;make-c1();f = c1.b1;co.p(f);

    make-c2();f = c2.b2;co.p(f);

    f = null;

   };
make-c1(){;+c1 == null{;c1 = new;

                         make-statics-c1();init-statics-c1();};};

make-statics-c1(){;c1.+b1:bool;};
init-statics-c1(){;make-c2();f = c2.b2;f = !f;

                   c1.b1 = f;f = null;};

make-c2(){;+c2 == null{;c2 = new;

                         make-statics-c2();init-statics-c2();};};

make-statics-c2(){;c2.+b2:bool;};
init-statics-c2(){;make-c1();f = c1.b1;f = !f;

                   c2.b2 = f;f = null;}
```

(with `f = !f` abbreviating `+f == false{;f = true;}{;f = false;}`). In addition with the convention that a boolean field will be initialized as `false` in molecular programming, this projection explains the result of the Java program. The corresponding molecule is depicted in Fig. 13.

The phenomenon of unexpected outcome of initializations can be phrased in terms of sensitivity for declaration orders. By simply changing the order in which classes are constructed one may obtain different output. In the following program class `c2` is constructed prior to class `c1`:

```
class c {

    static void m(){

    boolean b = c2.b2;

    co.p(c1.b1);

    co.p(c2.b2);

  }
}
```

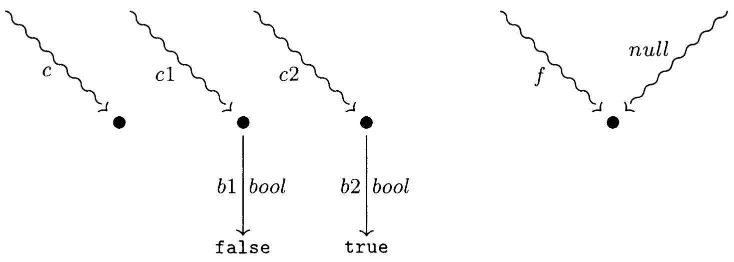

Fig. 13.

```
class c1 {
    static boolean b1 = !c2.b2;
    }
class c2 {
    static boolean b2 = !c1.b1;
}
```

Having understood the previous projection the output of this program is not surprising:

```
true
false
```

Our third program

```
class c {
static void m(){
    co.p(c1.b1);
    co.p(c2.b2);
    co.p(c1.d1);
    co.p(c2.d2);
  }
}
class c1 {
    static boolean b1 = c2.d2;
    static boolean d1 = true;
}
class c2 {
    static boolean d2 = true;
    static boolean b2 = c1.d1;
}
```

produces

```
true
false
true
true
```

Here the reason for this remarkable output is of a slightly different nature. The detailed analysis in terms of projections yields:

```
make-c();c.m();!;
make-c(){;+c == null{;c = new;
                      make-statics-c();init-statics-c();};};
make-statics-c(){;};
init-statics-c(){;};
m(){;make-c1();f = c1.b1;co.p(f);
     make-c2();f = c2.b2;co.p(f);
```

```
      make-c1();f = c1.d1;co.p(f);
      make-c2();f = c2.d2;co.p(f);
      f = null;
    };
make-c1(){;+c1 == null{;c1 = new;
                         make-statics-c1();init-statics-c1();};};
make-statics-c1(){;c1.+b1:bool;c1.+d1:bool;};
init-statics-c1(){;make-c2();f = c2.d2;c1.b1 = f;
                    f = true;c1.d1 = f;f = null;};
make-c2(){;+c2 == null{;c2 = new;
                         make-statics-c2();init-statics-c2();};};
                         make-statics-c2(){;c2.+d2:bool;c2.+b2:bool;};
init-statics-c2(){;f = true;c2.d2 = f;make-c1();
                    f = c1.d1;c2.b2 = f;f = null;}
```

This shows that the creation of the classes `c1` and `c2` are intertwined in such a way that initialization of `c2.b2` happens at a moment when `c1.d1` still has the default value `false`.

10. Conclusion

In this paper it is shown that molecular dynamics is able to model with fairly primitive means object-oriented programming techniques. The authors are convinced that this low-level and almost naive approach gives a clear insight into the nature of a whole range of aspects which are common to all kinds of object-oriented programming dialects.

References

[1] J.A. Bergstra, I. Bethke, P. Rodenburg, A propositional logic with 4 values: true, false, divergent and meaningless, J. Appl. Non-classical Logics 5 (2) (1995) 199–218.
[2] J.A. Bergstra, M.E. Loots, Program algebra for component code, Formal Aspects Comput. 12 (2000) 1–17.
[3] J.A. Bergstra, M.E. Loots, Program algebra for sequential code, J. Logic Algebr. Programming 51 (2002) 125–156.
[4] B. Diertens, The PGA—ProGramAlgebra website. Available from http://www.science.uva.nl/research/prog/projects/pga.
[5] S.C. Kleene, On a notation for ordinal numbers, J. Symbolic Logic 3 (1938) 150–155.
[6] D. Flanagan, P. Ferguson (Eds.), Java in a Nutshell: A Desktop Quick Reference, O'Reilly & Associates, Incorporated, 1999.